VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky

Diplomová práca

2009 Milan Pohančeník

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky

Refaktoring architektúry systému
Refactoring of the system architecture

2009 Milan Pohančeník

Ďakujem najmä svojmu vedúcemu diplomovej práce Ing. Štěpánovi Kuchařovi za ochotu a pomoc pri implementácii a písaní mojej práce, ďalej firme Crux Information Technology, pracovníkom firmy, svojím blízkym a ľuďom, ktorí rešpektujú a uznávajú moju prácu. Zároveň i všetkým ostatným ľuďom, ktorí ma podporujú a motivujú.

Prehlasujem, že som túto diplomovú práci vypracoval samostatne. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

V Ostrave …… ……..……..

Abstrakt

Hlavným účelom diplomovej práce bolo aplikovanie refaktoringu na zlepšenie kódu existujúceho systému. Samotný systém poskytuje dobré riešenie problematiky na ktorú je zameraný, no interná štruktúra sťažovala programátorom ich prácu a prinášala potom nové problémy. Niektoré moduly boli zasiahnuté i výkonovými problémami, ktoré refaktoring odhalil podľa predpokladov. Práca poskytuje teoretický základ o refaktoringu. Základ však nie je rozvíjaný o detaily, ale poskytuje príklady, ktoré vychádzajú z praktickej skúsenosti získanej počas refaktoringu systému. Záverečná časť je venovaná zjednodušenému predstaveniu systému a jeho modulov, architektúry a riešení, ktoré poskytuje. Riešenie problémov architektúry je predstavené na základnom a najobsiahlejšom module. Sú zároveň popísané i možnosti do budúcnosti, ktoré by systém spravili flexibilnejším a ľahšie rozšíriteľným.

Kľúčové slová

IT, ITIL, refaktoring, architektúra systému, webová aplikácia, návrhový vzor, java, softwarový proces

Abstract

The main goal of the diploma thesis is the practical application of refactoring to improve the code of existing system. The system itself offers good solution of the problems for which it is dedicated, but the internal structure of the code complicated the work of developers and caused new problems. Some modules were also affected by performance problems, which were identified by refactoring as supposed. The work presents the theoretic base of refactoring. It shows examples based on the practical experience gained from system refactoring and not details about refactoring itself. The end of this work is simply presenting the system, its modules, architecture and solutions which it offers. The solution of architecture problems is presented on the base and most important module. Possibilities of future improvement to the system, which could make the system more flexible and expandable, are also mentioned.

Key words

IT, ITIL, refactoring, system architecture, web application, design pattern, java, software process

Použité symboly a skratky

CMDB

CRM

CVS – Concurrent Versions System – systém pre správu verzií

HRE

HQL – jazyk pre popis dotazov v Hibernate

IT – informačné technológie

JPA – Java Persistence API, obdoba Hibernate technológie od firmy SUN.

JPQL – jazyk pre popis dotazov v JPA

JSF – Java Server Faces, framework pre prezentačnú vrstvu webových aplikácii

MVC

RFC – request for change

SLM

SVN – subversion, systém vychádzajúci z CVS, zdokonalený o niektoré funkcie

SOA

SOAP

XML

XP

Obsah

Úvod

K pojmu refaktoring existuje v dnešnej dobe množstvo kníh, webových stránok a zdrojov, ktoré obsahujú súbor najčastejších pravidiel používaných pri zlepšovaní kvality kódu, a tým aj aplikácie. Úlohou tejto práce nie je znovu popisovať už dostatočne definované pravidlá, či definovať znovu pojem refaktoring, ale hlavne uviesť praktický príklad refaktoringu na existujúcom kóde. Zároveň popisuje aj kompletnú zmenu architektúry existujúceho systému na novšie a modernejšie technológie. Pri tomto procese sa vyskytuje niekoľko základných problémov, ktoré je treba riešiť.

Diplomová práca je rozdelená na niekoľko nosných častí. V prvom rade je to popis a zoznámenie s refaktoringom a jeho metódami, výhodami a nevýhodami. Táto časť je doplnená praktickými príkladmi. Okrem iného sú zvažované výhodnosť a nevýhodnosť refaktoringu, či integrácia so softwarovým procesom. Ďalšia časť je venovaná refaktoringu reálneho systému. Systém je predstavený a je popísaná jeho súčasná architektúra. Je uvedený príklad najrozsiahlejšieho modulu, ktorý mal byť prevedený refaktoringom a optimalizovaný ako prvý. Nasleduje ešte krátke predstavenie zmien, ktoré by systému prospeli a mali by byť prevedené.

1. Refaktoring

V dnešnej dobe existuje mnoho spoločností, zoberajúcich sa vývojom softwaru. Každá z nich sa snaží svoj produkt udržiavať a rozvíjať ďalej podľa potrieb zákazníkov, či z vlastnej iniciatívy. Tento proces však prináša so sebou mnohé riziká. Ak v spoločnosti nie je daný žiadny postup (proces) pre správu produktov a ich vývoj, prípadne firma nemá dostatočne vzdelaných programátorov, môže dôjsť k postupnému úpadku kvality softwarového produktu. Toto sa časom prejaví v značnej miere na spravovateľnosti a ďalšom vývoji. Ten sa vo veľkej miere začne stávať príliš nákladným. Rovnako sa vyskytnú aj veľké množstvá chýb, ktoré sa budú odhaľovať veľmi ťažko a budú finančne náročné na opravu. Aby vôbec nedošlo k takejto hraničnej situácii, je potrebné priebežne kontrolovať a zlepšovať kvalitu kódu. Pre tento účel sa používa metóda tzv. refaktoringu. Samotná metóda však nestačí a je potrebné aj vyhradiť určité prostriedky na jej prevádzanie. Okrem toho je potrebné mať definovanú aspoň základnú skupinu pravidiel pre vývoj softwaru vo firme.

1.2 Definícia a význam

Samotné slovo refaktoring a jeho význam je popísané v niekoľkých literatúrach a zdrojoch. Refaktoring je možné chápať ako:

a. *„Zmena vnútornej štruktúry kódu za účelom ľahšieho pochopenia a ľahšej modifikácie.“* (1)
b. *„Reštrukturalizácia softwaru aplikáciou série refaktorovaní bez zmeny vonkajšieho chovania.“* (1)
c. *„Refaktorovanie je proces prevádzania zmien v softwarovom systéme takým spôsobom, že nemajú vplyv na vonkajšie chovanie kódu, ale vylepšujú jeho vnútornú štruktúru. Je to disciplinovaný spôsob prečisťovania kódu s minimálnym rizikom vnášania chýb.“* (2)

Všetky popisy slova refaktoring vyjadrujú jeho podstatu správne. Neexistuje žiadna presná definícia. Refaktoring je v skutočnosti proces, kedy programátori prechádzajú už vytvorený kód a píšu ho lepšie a kvalitnejšie. Nie je však nutné, aby ho vykonávali neustále, ale najvhodnejšie je, aby refaktoring aspoň raz za určitú dobu bol prevedený. Nie je však nutné prechádzať celý kód v aplikácii, ale postačí iba časť, s ktorou programátor momentálne má pracovať, prípadne niekoľko ďalších ovplyvnených tried. Užitočným sa môže stať pri identifikácii nevyužitých premenných, zbytočných cyklov, prípadne i zistíme chyby softwaru bez toho, aby sme ho testovali. Ďalšou fázou po refaktoringu môže byť optimalizácia výkonu.

Veľký význam má refaktoring pri veľkých projektoch, do ktorých je dopĺňaná neustále nová funkcionalita. Týmto sa mení analýza a teda aj celý návrh aplikácie a je potrebný refaktoring niektorých častí aby ostal kód čitateľný a udržiavateľný. Rovnako veľké projekty nemajú len jednu verziu, ale mnoho verzií. Z praktických skúseností je refaktoring vhodné použiť napr. po fázach projektu, kedy nebol časový priestor na udržiavanie čistého kódu a teda mnoho vecí bolo implementovaných ako náhradné riešenie. V takomto prípade by nahromadenie množstva nekvalitného kódu viedlo k znehodnoteniu systému, zníženiu jeho kvality a zvýšeniu finančných nákladov na údržbu a nový vývoj. Často sú však dopady nekvalitného kódu podceňované. Vývoj softwaru je skracovaný čo najviac, aby bolo možné hotový produkt predávať. Časom sa však nedá vyhnúť buď náročnému refaktoringu alebo úplnému stroskotaniu projektu. V druhom prípade je už len možný vývoj produktu od jeho počiatku. Toto stojí nemalé prostriedky, prípadne aj často končí stratou zákazníkov. Často vychádza potom pri veľkých projektoch investícia do refaktoringu menej ako nekontrolované programovanie.

1.3 Princípy refaktoringu

Refaktoring nie je súčasťou žiadnej fázy vývoja softwaru. Mal by byť vykonávaný priebežne spolu s fázou implementácie. Je to, ako bolo spomenuté *„zlepšenie existujúceho kódu“*. Zlepšenie kódu však prinesie často krát už len premenovanie premennej na názov, ktorý vystihuje jej funkciu, prípadne zmeniť množstvo premenných, umiestniť ich výhodnejším spôsobom. Hlavný problém býva aj často krát mnohonásobné využitie premennej s jedným názvom, čo vedie k zmäteniu programátora, ktorý následne takýto kód dostane.

Problémom pri čítaní kódu bývajú aj zložité vzorce pri výpočtoch, prípadne časti, kedy sa prevádzajú dáta do iného formátu (typický príklad je prevod z užívateľského formátu času na SQL čas / timestamp). Tieto konštrukcie bývajú veľmi komplikované, no ich rozdelenie na menšie časti a vhodné premenovanie premenných dokáže sprehľadniť tieto výpočty a prevody.

Ďalej je možné chápať refaktoring ako písanie kódu podľa správnych programovacích techník a vzoru objektovo orientovaného programovania (v prípade moderných objektovo orientovaných jazykov). Tomuto odpovedá napr. správny návrh rozhraní, tried, dedičnosti, polymorfizmu a iných výhod, ktoré tieto jazyky poskytujú. Zároveň je napr. samozrejmé využitie už vstavaných knižníc, ktoré tieto jazyky poskytujú štandardne. Toto nemusí byť nutne prípad, pokiaľ ide o starší systém, ktorý ma bežať na určitej verzii serveru, kde sa treba starať aj o prípadnú spätnú kompatibilitu.

Ide však nielen o čistotu z pohľadu návrhu, ale aj dodržiavanie istých zásad slušného programovania. Jednou z nich je aj dĺžka tela metód / funkcií v kóde. Dlhé telo metódy je veľmi nečitateľné a môže viesť k problémom pri údržbe, prípadne neskoršej úprave kódu. Takúto metódu nie je isto možné vhodne a stručne okomentovať a popísať činnosť na jednotlivých riadkoch. Ďalším zlým príkladom programovania je používanie pevne zadaných hodnôt, ktoré sa potom vyskytujú na rôznych miestach kódu. V prípade potreby zmeny takejto hodnoty z akéhokoľvek dôvodu toto nie je možné spraviť pri veľkých projektoch. Preto je vhodné zavádzať konštanty, do ktorých sú hodnoty uložené a tieto následne používať. Používanie konštánt v podmienkach a príkazoch typu switch zvyšuje prehľadnosť činnosti metódy.

V konečnom dôsledku refaktoring zlého kódu vždy šetrí čas programátorov pri neskoršej údržbe, či rozšírení funkcionality aplikácie. To umožňuje rýchlejšie vyvíjať a opravovať a tak šetriť finančné zdroje. Hlavným princípom je teda čistota kódu. Finančné prostriedky vynaložené na prevedenie refaktoringu sa vrátia v neskorších fázach projektu.

1.3.1 Vyňatie kódu a nahradenie metódou

Metóda je najjednoduchšou formou refaktoringu, s akou je možné sa stretnúť. Bežne sa používa bez toho, aby si bol programátor vedomý, že prevádza refaktoring. Jej princíp a použitie je veľmi jednoduché, no efektívne. Ide len o jednoduché prehodnotenie kódu vnútri vytvorenej metódy / funkcie a rozdelenie na menšie, logicky viazané celky. Ideálne je rozdeliť kód na celky podľa toho, akú funkciu vykonávajú. Ak je možné niektoré zlúčiť do jednej metódy, je to určite správny postup. Na príklade ZK. 1 je možné tento postup sledovať detailne na jednoduchom kóde v jazyku java. Prvý kód je čistý a bez akéhokoľvek refaktoringu.

```
int[] numbers = new int[args.length];
for (int i=0; i<args.length; i++) {
  numbers[i] = Integer.parseInt(args[i]);
}
int result = 0;
for (int i=0; i<numbers.length; i++) {
  if(i % 2 != 0)
    result++;
}
if ( result > 0 )
  System.out.println("Mnozina obsahuje aspon jedno neparne cislo");
else
  System.out.println("Mnozina neobsahuje ani jedno neparne cislo");
```

ZK. 1 KÓD BEZ REFAKTORINGU

Ako je vidno, v kóde sa nachádzajú dva cykly, ktoré prevádzajú jednoduché operácie a potom výpisy, ktoré sa starajú o informáciu užívateľa. Bez komentára k danému kódu nie je ihneď vidieť, čo dané cykly robia. V prvom prípade ide o jednoduché načítanie hodnôt zo vstupu a ich prevedenie na pole celočíselných hodnôt. V druhom prípade je však komplikovanejšie rozpoznať o akú činnosť sa jedná. Analýzou sa dá zistiť, že kód počíta iba nepárne čísla a výsledok vráti do lokálnej premennej *result*. Tieto dva cykly je možné napísať prehľadnejšie a to napr. ako je vidno na príkladoch ZK. 2 a ZK. 3.

```
private static int countOdd(int[] numbers) {
  int result = 0;
  for (int i=0; i<numbers.length; i++) {
    if(i % 2 != 0)
      result++;
  }
  return result;
}

private static int[] parseInput(String[] args) {
  int[] numbers = new int[args.length];
  for (int i=0; i<args.length; i++) {
    numbers[i] = Integer.parseInt(args[i]);
  }
  return numbers;
}
...
int[] numbers = parseInput(args);
int result = countOdd(numbers);
if ( result > 0 )
  System.out.println("Mnozina obsahuje aspon jedno neparne cislo");
else
  System.out.println("Mnozina neobsahuje ani jedno neparne cislo");
...
```

ZK. 2 KÓD PO PRVOM REFAKTORINGU

V tomto prípade bol refaktoring prevedený vyňatím dvoch cyklov a umiestnením do samostatných metód. Ich návratová hodnota je uložená v lokálnych premenných a ako parametre vstupujú pôvodné lokálne premenné kódu. Rozložený kód je takto čitateľnejší. Zo samotného názvu metód je možné posúdiť, akú funkcionalitu plnia. Refaktorovanie je možné previesť touto metódou ešte ďalej. Na odstránenie poslednej časti je možné znovu použiť vyňatie kódu a nahradenie metódou. Kód po úprave

je na príklade ZK. 3. Výpis je nahradený volaním metódy s parametrom, ktorá nič nevracia. Takto upravený program je ľahšie čitateľný a jeho funkcionalita sa ľahko dá zmeniť. Nahradením metód je možné prepísať program napr. na počítanie maxima, minima, či ďalších parametrov.

```
private static void printResult(int result) {
  if ( result > 0 )
    System.out.println("Mnozina obsahuje aspon jedno neparne cislo");
  else
    System.out.println("Mnozina neobsahuje ani jedno neparne cislo");
}
...
printResult(result);
...
```

ZK. 3 POSLEDNÁ ÚPRAVA KÓDU

Na tomto kóde je ešte možné previesť jednu úpravu, ktorou je premenovanie lokálnej premennej *result* napr. na *oddCount*, ktorá bude lepšie vyjadrovať jej presný význam.

Opakom tejto metódy je veľmi jednoduchá metóda *zlúčenie metód*. Táto miesto toho, aby programátor rozdeľoval kód na viacero blokov, vynímal osobitné celky, tak robí opačnú činnosť. Je to možné použiť v prípade zbytočných metód, ktorých funkcia je veľmi jednoduchá a jej funkciu je možné priamo odvodiť z obsahu. Medzi takéto metódy patria aj tie, ktoré sú volané iba na jednom mieste programu. Vtedy zbytočne zaberajú miesto v kóde a nie je možné sa vyznať v množstve metód.

1.3.2 Nahradenie hodnoty konštantou

Táto metóda je veľmi jednoduchá a efektívna pri úprave existujúceho kódu. Jej význam je možné oceniť pri aplikáciách, ktoré využívajú množstvo hodnôt napr. pri porovnávaní v podmienkach. Ďalšie využitie konštánt je pri inicializácii premenných vnútri objektov, prípadne návratové hodnoty a mnoho ďalších. Miesto čísiel, ktoré nedokážu vysvetliť svoju podstatu, je možné nahradiť konštantu s vhodným názvom. Všetky výskyty hodnoty musia byť potom nahradené konštantou. Využitie tejto metódy sa uplatňuje veľmi často vtedy, ak hodnota z jedenej triedy je ďalej využívaná v iných triedach. Pokiaľ k takejto situácii dôjde, je nutné ihneď nahradiť hodnotu konštantou, a to v triede, z ktorej konštanta pochádza, prípadne patrí. Pokiaľ to nie je jednoznačne rozhodnuteľné, je nutné prehodnotiť štruktúru tried a zaviesť novú triedu, prípadne rozhranie, kde budú tieto konštanty uložené.

1.3.3 Zavedenie vysvetľujúcej premennej

Pri programovaní najmä matematicky založených programov (grafika, fyzikálne výpočty, výpočty účtovné, ...) sa vyskytuje často krát formát kódu, ktorý nemá žiadnu vysvetľovaciu hodnotu. Programátor, ktorý následne takýto kód dostane, nemá potom možnosť pochopiť napr. aký vzorec bol použitý, prípadne aký význam majú jednotlivé výpočty. Odhaľovanie chyby v takomto kóde je náročné a zdĺhavé a teda i z finančnej stránky drahé. Typický príklad výpočtu nájdeme na ZK. 4. Je to najčastejšie používaný príklad na túto metódu. Jedná sa o výpočet ceny za objednávku pre zákazníka. Zákazník, ktorý je registrovaný má zľavu 5%, zákazník, ktorý nie je registrovaný nemá žiadnu zľavu a platí špeciálna zľava pre registrovaných zákazníkov, ktorí majú objednávku nad určitú sumu. Hraničná suma je 10 000kč a zľava 10%. Navyše je k cene pripočítané i poštovné, pokiaľ objednávka pred zľavou nedosiahla aspoň 5001 Kč.

```
public double getFinalPrice() {
  double price =
      basket.getPrice() - basket.getPrice() *
      (isRegistered()?(basket.getPrice()>10000?0.10:0.05):0) +
      (basket.getPrice() > 5000 ? 0 : 150);
  return price;
}
```

ZK. 4 VÝPOČET CENY (NEREFAKTOROVANÉ)

Ako je vidno z príkladu ZK. 4, výpočet nie je vôbec okomentovaný a to, že cena je počítaná istým systémom so zľavami a prideleným poštovným je možné sa len dočítať z textu a potom ho nájsť vo vzorci. Takýto formát by sa v kóde nemal vôbec vyskytovať. Dôvod je stále rovnaký a to čitateľnosť kódu a následné úpravy v ňom. Tento problém je však veľmi jednoducho riešiteľný, a to rozložením veľkého výpočtu na čiastkové malé logické celky a potom výpočet celkovej ceny.

```
public double getFinalPrice() {
  double registeredDiscount =
      basket.getPrice() * (basket.getPrice() > 10000 ? 0.10 : 0.05);
  double postCosts = basket.getPrice() > 5000 ? 0 : 150;
  double basketPrice = basket.getPrice() - registeredDiscount;
  double fullPrice = basketPrice + postCosts;
  return fullPrice;
}
```

ZK. 5 VÝPOČET CENY (REFAKTOROVANÉ)

Jedno priradenie do premennej bolo rozdelené na viac častí. Každá časť vzorca tvorí logický celok pri výpočte a má svoju premennú, ktorá má dostatočne výstižný názov. Z takto upraveného kódu je ľahké vyčítať jednotlivé časti vzorca a uvedomiť si, kde je možná chyba, prípadne ako fungujú. Celkový výpočet je priradený do poslednej premennej a vrátený ako výsledok. Samozrejmosťou je ešte dodatočné nahradenie pevných hodnôt buď konštantami alebo v prípade e-shopov premennými načítanými z databázy. V tomto prípade sa hodnoty môžu meniť a pevné zadanie by bolo nepraktické.

1.3.4 Nahradenie zoznamu parametrov objektom

Často sa pri programovaní aplikácií stáva to, že metódy mávajú pri sebe veľmi dlhý zoznam parametrov. Ich volania potom bývajú veľmi dlhé. Aby sa tomuto predišlo, je nutné parametre buď odstrániť, alebo zaviesť objekt, ktorý parametre ponesie. Samozrejme, niekedy nie je možné túto formu refaktoringu previesť dokonale, keďže parametre je dobré rozdeľovať podľa ich logických významov a zaviesť objekt s výstižným názvom. Praktická ukážka ako môže vyzerať nesprávne napísaná metóda s veľkým počtom parametrov je na ukážke ZK. 6. Táto metóda môže počítať dátum, do ktorého je potrebné reagovať na vyriešenie problému (príklad ITIL softwaru, správa incidentov, ...). Metóda ma zavedených mnoho parametrov, no jednoduchým pohľadom je možné zistiť, že tieto zviazané a vyjadrujú dni v týždni a napr. pracovnú dobu zamestnanca, prípadne dobu behu aplikácie. Pre všetky tieto parametre je možné zaviesť jeden parameter a zoskupiť ich do objektu s názvom napr. *WorkingHours* v prípade zamestnanca a jeho pracovnej doby. Tento objekt potom bude mať jednotlivé parametre nastavené ako svoje vlastnosti a bude ich navonok sprístupňovať pomocou verejných metód. V prípade, že vlastnosť je daná napr. konštantou a je nemenná, potom nebude mať objekt metódu, ktorou by vlastnosť nastavil. Príklad takéhoto objektu je na ZK. 7.

```
public Date countResponseTime
      (Date mondayFrom, Date mondayTo, Date tuesdayFrom, Date tuesdayTo,
      Date wednesdayFrom, Date wednesdayTo, Date thursdayFrom, Date
      thursdayTo, Date fridayFrom, Date fridayTo)
{
  ...
}
```

ZK. 6 POČÍTANIE REAKČNÉHO ČASU (NEREFAKTOROVANÉ)

Objekt obsahuje presne tie vlastnosti, ktoré mala pôvodná metóda ako svoje parametre. Všetky vlastnosti musíme vedieť nastaviť v prípade, že ich chceme načítať napr. z webového formulára. Teda objekt sprostredkúva nastavovacie metódy pre svoje vlastnosti. Zároveň potrebujeme, aby sme mohli každú vlastnosť čítať, teda sprostredkúva metódy na získanie hodnoty. Tieto budú potom použité miesto výskytov pôvodných hodnôt. Napr. ak v pôvodnom kóde bol použitý parameter *mondayFrom*, potom tento nahradíme volaním *[objekt].getMondayFrom()*. Kde *[object]* je konkrétny objekt, ktorý bol vytvorený predtým za pomoci konštruktoru a naplnený správnymi hodnotami.

```
public class WorkingHours {
      private Date mondayFrom;
      private Date mondayTo;
      ...
      public WorkingHours(Date mondayFrom, Date mondayTo, ...) {
            this.mondayFrom = mondayFrom;
            this.mondayTo = mondayTo;
            ...
      }

      public Date getMondayFrom() {
            return mondayFrom;
      }

      public Date setMondayTo() {
            return mondayTo;
      }
      ...
}
```

ZK. 7 OBJEKT PRE PARAMETRE

Ukážkový príklad neobsahuje všetky vlastnosti a objekty kvôli prehľadnosti. Je však jasne vidno princíp, akým sú parametre predané nosnému objektu. Objekt môže byť vytvorený pomocou štandardného konštruktora a napĺňaný potom pomocou „set“ metód. Takýto postup je vhodný napr. pri postupnom parsovaní a napĺňaní dát. Vytvoriť objekt parametrov je možné teda napr. použitím definovaného konštruktoru podľa príkladu ZK. 8 alebo podľa príkladu ZK. 9. Oboje sú správne, je iba na programátorovi, ktoré z nich použije.

```
WorkingHours hours = new WorkingHours(mondayFrom, mondayTo, ...);
```

ZK. 8 PRVÁ METÓDA VYTVORENIA OBJEKTU

Výhodou tohto prístupu k programovaniu a zavádzania objektov miesto parametrov je určite lepšie rozloženie kódu v aplikácii a jej flexibilita pri neskoršom programovaní a úpravách. Pri napĺňaní objektu zavedeného miesto parametrov nie je vidno veľkú výhodu v zavedení tohto prístupu. Avšak to, že objekt je potrebné plniť a najprv spracovať všetky údaje, ktoré potrebujeme, je možné

previesť napr. na adaptér, ktorý sa o spracovanie a naplnenie objektu postará. Takto je možné ušetriť písanie kódu. Pri použití ďalších metód je možné ešte kód upraviť a vo finálnej podobe tak dosiahnuť ľahko modifikovateľného programu.

```
WorkingHours hours = new WorkingHours();
hours.setMondayFrom(mondayFrom);
hours.setMondayTo(mondayTo);
...
```

ZK. 9 DRUHÁ METÓDA VYTVORENIA OBJEKTU

1.3.5 Vytvorenie dedičnosti z objektov

V návrhu aplikácie sa často môžu vyskytnúť zmeny, prípadne software ako taký sa vyvíja. Nová funkcionalita musí byť však implementovaná tak, aby neporušovala základné programátorské techniky a princípy objektovo orientovaného programovania. V opačnom prípade je nutné previesť refaktoring, pretože výsledný kód sa môže stať pre neskoršie úpravy nepoužiteľný, prípadne refaktoring bude aj tak nutné previesť. Finančné zdroje vynaložené na neskoršie úpravy budú omnoho vyššie ako zdroje vynaložené pri refaktoringu prevádzanom s novou implementáciou funkcionality. Príkladom novej funkcionality môže byť rozdelenie napr. služieb v ITIL systéme na služby technické a obchodné. Pôvodná hierarchia s takýmto systémom nepočítala a vytvorenie dvoch nových služieb môže byť prevedené napr. podľa príkladu ZK. 10.

```
public class Service {
  public static final int BUSINESS_TYPE = 0;
  public static final int TECHNICAL_TYPE = 1;
  private int type;
  ...
  public int getType() {
      return type;
  }

  public void setType(int type) {
      this.type = type;
  }
  ...
  public void save() {
    switch (getType()) {
      case BUSINESS_TYPE:
      DBFactory.getServiceDB().saveBusinessServiceHeader(this);
      break;
      case TECHNICAL_TYPE:
      DBFactory.getServiceDB().saveTechnicalServiceHeader(this);
      break;
    }
    DBFactory.getServiceDB().save(this);
  }
  ...
}
```

ZK. 10 SLUŽBA TECHNICKÁ A OBCHODNÁ

Ak bude implementácia prevedená takto, je určite viditeľné, že pravdepodobne z dôsledku nedostatku času pri implementácii programátor ponechal pôvodnú triedu na služby a iba doplnil nové vlastnosti a upravil metódy pre uloženie rozhodovacím blokom. Toto spôsobí neskôr veľké problémy,

pretože nebude možné odlíšiť, ktoré vlastnosti sú špecifické pre technickú a ktoré pre obchodnú službu. Zároveň v kóde bude nutné ošetrovať typy služieb priebežne, čiže takýchto rozhodovacích blokov bude výsledný software obsahovať niekoľko a kvalita softwarového diela začne upadať. Začne sa objavovať množstvo chýb v dôsledku použitia rozhodovacích blokov. Ak niektorý z programátorov zle pochopí štruktúru a rozdiely medzi jednotlivými typmi služieb, nastane ešte viac problémov. Tieto budú ťažko odstrániteľné a povedú buď k refaktoringu alebo k zavrhnutiu celého systému a nového vývoja. Dôsledky budú každopádne veľké.

```
public abstract class Service {
  public static final int BUSINESS_TYPE = 0;
  public static final int TECHNICAL_TYPE = 1;
  protected int type;

  public int getType() {
      return type;
  }

  public void setType(int type) {
      this.type = type;
  }

  public abstract void save();
}
```

ZK. 11 REFAKTOROVANÁ TRIEDA SERVICE

Lepší prístup je založený na dedičnosti objektov. V tomto prípade to bude dedičnosť služieb. Určite bude existovať trieda s názvom *Service*, ktorá bude rodičovskou triedou. V nej budú koncentrované vlastnosti, metódy a konštruktory, ktoré sú spoločné ako pre technickú, tak i obchodnú službu. Ďalej bude obsahovať abstraktné metódy, ktoré sú síce spoločné pre dediace triedy, ale ich výsledná implementácia je rozdielna a nie je možné ich spojiť a vytvoriť v nadtriede. To, že trieda bude obsahovať abstraktné metódy automaticky znamená, že bude musieť aj samotná trieda byť deklarovaná ako abstraktná. Z triedy *Service* sa teda stane abstraktná trieda a bude obsahovať všetky metódy spoločné pre všetky typy služieb. Okrem toho bude obsahovať aj abstraktné metódy, ktoré budú dediace triedy implementovať. Za povšimnutie stojí i to, že vlastnosť *type* v triede *Service* zmenila svoju viditeľnosť z *private* na *protected*. Znamená to, že dediace triedy k nej môžu taktiež pristupovať. Táto vlastnosť je využitá potom v konštruktoroch dediacich tried. Každá nastaví svoj presný typ, ktorý je potom dostupný počas životnosti objektu. V ostatných situáciách, keď by bolo potrebné využiť typ služby, je možné toto zistiť jednoducho cez metódu *getType()* priamo z abstraktnej triedy. Trieda potom bude vyzerať približne podľa príkladu ZK. 11.

```
public class BusinessService extends Service{
  public BusinessService() {
      this.type = BUSINESS_TYPE;
  }
  ...
  @Override
  public void save() {
      DBFactoryRefactored.getServiceDB().saveBusinessServiceHeader(this);
      DBFactoryRefactored.getServiceDB().save(this);
  }
}
```

ZK. 12 TRIEDA OBCHODNEJ SLUŽBY

Triedy dediace z rodičovskej triedy sú na príkladoch ZK. 12 a ZK. 13. Je možné vidieť jednoduchý systém akým je implementovaná potom každá metóda *save()* a konštruktor každej z tried. Pre uloženie služby stačí zavolať len napr. *[service].save(),* kde *[service]* je objekt, ktorý bol vytvorený predtým za použitia konštruktoru technickej alebo obchodnej služby a je typu *Service*, teda rodičovskej triedy. Ide len o príklady, a tak nie sú uvedené všetky možné vlastnosti a metódy, ale len časť dôležitá pre ilustráciu princípu. Možným refaktoringom v tejto štruktúre by bolo možno ešte vyňatie typu služby *type* a vytvorenie vlastného objektu napr. *ServiceType*, ktorý by obsahoval konštanty pre typy služieb a vedel by udržať, o aký typ služby ide. Táto časť je však len ďalším krokom. V reáli nejde o veľkú chybu a preto nie je potrebné zatiaľ ju prevádzať. Pokiaľ by však množstvo typov služieb narástlo napr. na desať, je treba zvážiť tento jednoduchý postup vyňatia triedy a zavedenie ako premennej do typu *Service*. Určite by to skrátilo dĺžku triedy pre služby a sprehľadnilo kód.

```
public class TechnicalService extends Service {
  public TechnicalService() {
      this.type = TECHNICAL_TYPE;
  }
  ...
  @Override
  public void save() {
    DBFactoryRefactored.getServiceDB().saveTechnicalServiceHeader(this);
    DBFactoryRefactored.getServiceDB().save(this);
  }
}
```

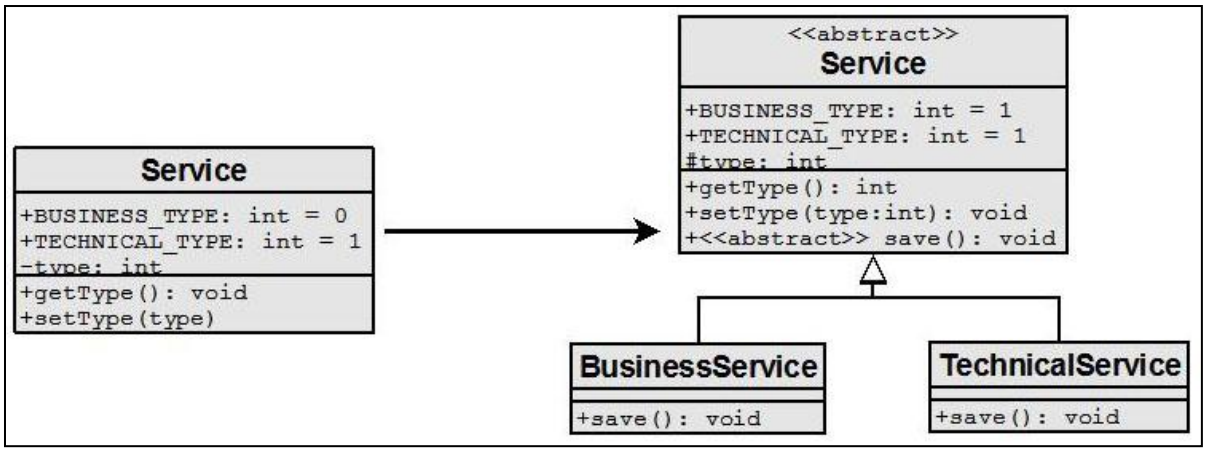
ZK. 13 TRIEDA TECHNICKEJ SLUŽBY

1.3.6 Náhrada podmienky polymorfizmom

Metóda núti využiť jednu z dôležitých vlastností objektovo orientovaného programovania – polymorfizmus. Jej prínos pre neskoršie úpravy a vývoj softwaru je značný. Príklad použitia je hlavne pri rozhodovacích blokoch, kedy sa na základe typu objektu rozhodujeme o vrátení inej hodnoty. Pokiaľ sa takéto bloky v softwari vyskytujú, mal by sa programátor zamyslieť nad možným prevodom na polymorfizmus.

Typickým príkladom je v predošlom odstavci metóda *save()*. Táto v pôvodnej podobe sa rozhodla na základe typu služby o vyvolaní akcií pre uloženie rôznych typov služieb. Rozhodovacie bloky však nie sú najvhodnejším spôsobom, ako implementovať takúto funkciu. Lepším je zavedenie dedičnosti objektov a využitie polymorfizmu pri rozhodnutí. Takto sa nie je nutné starať o správne rozhodnutie, ale toto za nás spraví prekladač. V každom prípade sa polymorfizmus bude chovať rovnako, čo je i žiaduce. V prípade ručnej implementácie sa môžu vyskytnúť nesprávne napísané bloky a viesť to k nepredpokladaným problémom a pádom systému.

Pri prevedení na polymorfizmus bola nahradená štruktúra jednej triedy dedičnosťou. Zmenu je možné vidieť na Obr. 1. Metóda, ktorá obsahovala rozhodovací blok je v nadtriede vedená ako *abstract*, čo zaručí, že dediace triedy musia túto metódu implementovať. Výsledkom je kód z príkladov ZK. 12 a ZK. 13. Obsahuje novo zavedenú dedičnosť a spolieha sa na polymorfizmus pri volaní metódy *save()*.

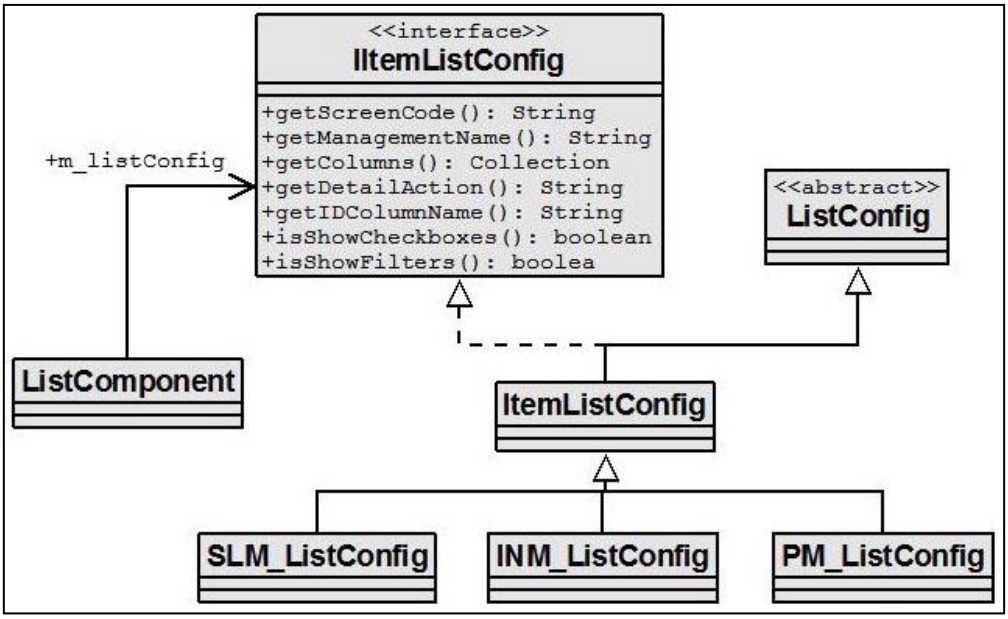


OBR. 1 DIAGRAM TRIED PRE SLUŽBY

1.3.7 Vytvorenie spoločného rozhrania pre triedy

Rozhrania fungujú podobným princípom, ako abstraktné triedy. Rozdielom oproti abstraktnej triede v jazyku Java je fakt, že rozhranie nesmie implementovať vôbec telo žiadnej metódy, ani definovať premenné, ku ktorým budú slúžiť prístupové *get* a *set* metódy. Navyše existuje možnosť implementácie mnohých rozhraní, ale nie viacnásobná dedičnosť. Do rozhraní sa väčšinou aj ukladajú konštanty, ktoré sa používajú v celom systéme. Samotné rozhranie slúži ako forma dohody, ktorá znamená, že implementujúce triedy budú mať daný zoznam spoločných metód, pomocou ktorých komunikujú s okolím. Takto je zavedená väčšia univerzálnosť pri používaní tried.

Niekedy je zložité určiť, či je lepšie použiť rozhranie alebo dedičnosť. Existuje však jednoduchý princíp, podľa ktorého je možné sa rozhodnúť. Ak je potrebné sa rozhodnúť o použití rozhrania alebo dedičnosti a metódy, ktoré majú mať triedy spoločné sú natoľko odlišné, že by boli všetky abstraktné, potom je isto výhodnejšie použiť rozhranie. Ak metódy budú mať spoločnú časť, prípadne existuje istý počet metód, ktoré budú dokonca úplne rovnaké, je lepšie použiť dedičnosť.



OBR. 2 TRIEDNY DIAGRAM PRE KONFIGURÁCIU LISTOV

Pekný príklad poskytuje implementácia listov v systéme *ServiceBase*. Listy sú využívané na zobrazovanie zoznamov objektov, umožňujú stránkovanie, triedenie a filtráciu položiek. O aký typ listu ide je možné rozhodnúť na základe konfiguračnej triedy. Každý typ listu má svoju vlastnú triedu, ktorá obsahuje napr. typ logiky, ktorá sa má volať pri vyberaní dát alebo definované stĺpce, ktoré má zobraziť. Každá táto trieda pritom implementuje spoločné rozhranie, pomocou ktorého komunikuje potom s logikou zobrazovania listov. Časť štruktúry je vidno na obrázku Obr. 2. Je uvedená iba časť štruktúry, ktorú majú listy. Všetky však majú spoločné rozhranie, ktorým potom komunikujú s triedou *ListComponent*, starajúcu sa o vykonávanie logiky listov.

1.3.8 Náhrada pole za objekt

Polia ako také sú hodne často využívané v programovaní. Umožňujú jednoduchú prácu a prístup k jeho obsahu. Využitie majú aj pri udržiavaní zoznamu hodnôt. Niekedy však dochádza k javu, že programátor využíva pole na uloženie rôznych hodnôt, ktoré by nemali byť spolu uložené v poli. Najjednoduchším príkladom je jazyk php a spôsob čítania dát z databázy. Vo výsledku dotazu nad databázou sa nachádzalo pole hodnôt, ktoré symbolizovali jednotlivé stĺpce databázy v poradí, v akom boli napísané v dotaze. Programátor pristupoval potom k jednotlivým hodnotám podľa poradia. Ak došlo k zmene v poradí stĺpcov, automaticky došlo k chybnému zobrazeniu stránky u klienta.

Postup vyššie by mal byť nahradený objektom, v ktorom bude každá jeho vlastnosť nastavená osobitne a mať svoje prístupové metódy. I keď pri zmene poradia stĺpcov v dotaze, ktorý vracia iba pole môže dôjsť k zámene, ale táto bude jednoducho odstrániteľná a volanie vlastností objektu zabezpečí, že nebudeme musieť meniť poradie na iných miestach

Aj keď chyba tohto typu sa často pri programovaní nevyskytuje, niekedy je ju možné nájsť najmä v starších zdrojových kódoch. Najmä programátori – začiatočníci sa dopúšťajú často tejto chyby. V jednoduchých systémoch toto neprekáža, no v systémoch, ktoré sú komerčné a počíta sa s rastom a vývojom sa nesmie takýto spôsob práce s dátami vyskytovať.

1.3.9 Zmena chybových kódov na výnimky

Je nie celkom presnou metódou refaktoringu. V starších aplikáciách programovaných hlavne pred vznikom výnimiek boli chybové stavy ošetrované kódmi chyby. Na základe kódu bolo neskôr rozhodnuté, aká časť bude ďalej prevedená a akým štýlom bude informácia podaná užívateľovi. V tomto systéme chybových kódov však je nevýhoda, keďže ošetriť návratovú hodnotu je potrebné ihneď po získaní výsledku z funkcie. Ak by bolo potrebné vrátiť chybu na vyššiu úroveň, je nutné zaviesť systém ako predať chybu. Toto však je možné lepšie previesť za použitia výnimiek, ktoré je možné odchytávať na potrebných úrovniach a vrátiť užívateľovi upozornenie, prípadne použiť výnimky iným spôsobom.

Majme metódu, ktorá zabezpečuje uloženie napr. nového incidentu do databázy. Ak je systém navrhnutý tak, že incident musí mať unikátne meno, potom je nutné, aby toto bolo ošetrené návratovým kódom. Tento môže byť napr. hodnota -2. Druhý návratový kód bude napr. -1, a to v prípade, že nastane iný typ chyby (prerušené databázové spojenie, zlé dátové typy, ...). Ak bude aplikácia spracovaná týmto spôsobom, potom bude zdrojový kód vyzerať napr. podľa príkladu ZK. 14.

```
public void saveIncident(incident) {
  int savedIncident = DBFactory.getIncidentManagement().save(incident);
  if(savedIncident == -1)
      System.out.println("Cannot save incident!");
  else if(savedIncident == -2)
      System.out.println("Duplicate incident name");
}
```

ZK. 14 OŠETRENIE CHÝB NÁVRATOVÝMI KÓDMI

Ak bude programátor chcieť zaviesť nové chybové kódy, je potrebné doplniť tieto do vetvy. Toto nie je veľký problém. Veľkým problémom v tomto prípade však bude zaviesť systém tak, aby bola chyba pre existujúci názov ošetrená na vyššej úrovni a databázová chyba spôsobila úplne iný výpis a iné chovanie aplikácie. Preto je výhodné použiť systém výnimiek, pokiaľ to daný programovací jazyk umožňuje. Z príkladu ZK. 15 je možné vidieť ako bude trieda po prevedení refaktoringu vyzerať. Je uvedená aj trieda, ktorá volá metódu *saveIncident*, aby bolo vidno, ako je možné ošetrovať chyby za pomoci výnimiek. Jazyk java má napr. výnimku *RuntimeException*, ktorá slúži ako identifikátor závažného problému. Na príklade je použitá v prípade, že nastala iná výnimka (napr. prerušené databázové spojenie,...) a v programe nie je možné pokračovať.

```
public void saveIncident(incident) throws DuplicateNameException {
  DBFactory.getIncidentManagement().save(incident);
}
```

ZK. 15 REFAKTOROVANÁ METÓDA SAVEINCIDENT()

Ako je možné vidieť, metóda *saveIncident* je po refaktoringu kratšia a boli odstránené podmienky, ktoré ošetrujú chybové kódy. Navyše pribudla výnimka *DuplicateNameException*, ktorá bude signalizovať, že meno incidentu už existuje. Výnimku deleguje metóda na vyššiu úroveň, kde sa bude ošetrovať. V príklade ZK. 16 je možné potom vidieť metódu, ktorá je hlavnou v celej aplikácii a stará sa o odchytávanie výnimiek. Je vidieť, že takýmto spôsobom je možné vytvoriť veľmi dômyselný systém, ktorý bude ľahké spravovať a prípadne doplniť o novú funkcionalitu. Zabezpečenie aplikácie pomocou chybových kódov by bolo omnoho náročnejšie a vyžadovalo viac zdrojov pre úpravu kódu.

```
public void execute() {
  try {
    ...
    saveIncident(incident);
  } catch (DuplicateNameException e) {
    ...
    e.getMessage();
  }
}
```

ZK. 16 METÓDA HLAVNEJ TRIEDY S VÝNIMKAMI

Samotná metóda databázovej vrstvy, ktorá sa stará o uloženie bude potom modifikovaná tak, že nebude vracať celočíselný kód o prebehnutej akcii, ale len vyhodí výnimku podľa konkrétnej chyby, ktorá nastala. Refaktorovaná metóda môže teda vyhodiť výnimku *DuplicateNameException*, prípadne *RuntimeException*, ak dôjde k inej chybe pri prebiehajúcom ukladaní. Samozrejme je nutné, aby pred vyvolaním výnimky bol zavolaný rollback na databázu. Pokiaľ aj tento bude zlyhať, potom bude vyhodená *RuntimeException*, čo bude znamenať vážnu chybu a s aplikáciou nebude možné ďalej pracovať. Refaktorovaná metóda *save()* je na príklade ZK. 17.

```
public void save() throws DuplicateNameException {
  if(nameExists)
    throw new DuplicateNameException("Duplicate incident name");
  if(error)
    throw new RuntimeException();
}
```

ZK. 17 REFAKTOROVANÁ METÓDA SAVE()

1.4 Návrh aplikácie a refaktoring

Návrh aplikácie je celkovo zložitý proces, ktorý však musí byť prevedený dôkladne s ohľadom na požiadavky projektu. Ak bude zanedbaná v projekte fáza návrhu, nebude mať aplikácia jednotnú architektúru, čo povedie k mnohým formám implementácie výsledného projektu a celej aplikácie. Samozrejmé je, že takýto prístup povedie k neskorším problémom so spravovateľnosťou a rozširovaním aplikácie. Časté chyby v aplikácii nebudú výnimkou a povedú buď k neúspešnému, prípadne predčasnému koncu projektu, alebo k navýšeniu finančných prostriedkov vynaložených na dokončenie aplikácie. Pokiaľ projekt postupuje týmto smerom, je nutné urýchlene jednať a zamyslieť sa nad upravením návrhu a postavením novej architektúry. Zmena nemusí viesť nutne k pozastaveniu doterajšieho vývoja a úplne novej implementácii. Nakoľko existujú metódy refaktoringu, je možné tieto použiť na existujúci projekt a takto zdokonaliť návrh aplikácie. Včasný refaktoring je pomerne jednoduchý a nezaberie veľa prostriedkov a zdrojov. Bude ľahšie previesť zmeny ako vo finálnych fázach projektu, kedy by už mali byť odstraňované len vyskytujúce sa chyby. I samotný výsledok refaktoringu bude mať väčší účinok.

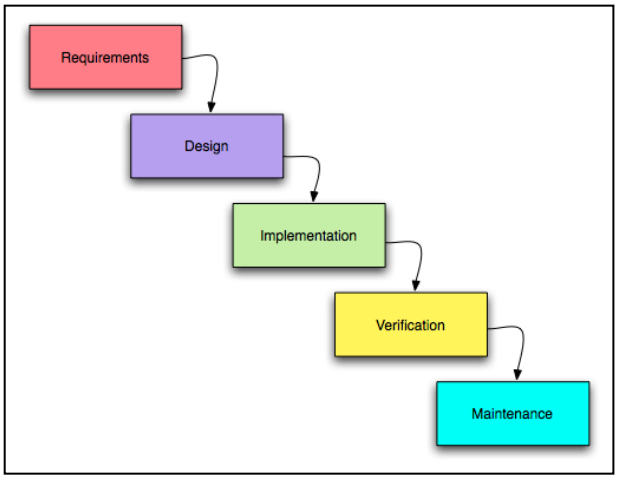
Spôsobené rozpory s návrhom aplikácie nemusel však spôsobiť len zlý návrh aplikácie v počiatočných fázach. K veľkým zmenám návrhu vedú často i nové požiadavky zákazníka. V prípade, že projekt je pre zákazníka nastavený tak, že môže požadovať dodatočnú funkcionalitu, je možné že navrhnutá architektúra nebude vyhovovať a bude ju nutné prehodnotiť a upraviť. Často nespôsobujú problémy len návrhy aplikácie, ale napr. i samotný dátový model, ktorý sa s novými požiadavkami musí zmeniť. Pre takéto prípady nie je iné možné riešenie ako použiť kvalitný refaktoring. Refaktorovať je nutné po fázach a nikdy nie všetky časti naraz. Takýto prístup by viedol k istému neúspechu najmä pri veľkých projektoch. Refaktorovanie po fázach prináša pomalší, ale bezpečný prístup. Po každej fáze je možné upraviť mierne návrh tak, aby sa blížil požiadavkám zmeny. Po ukončení refaktoringu bude návrh aplikácie a jej implementácia úplne iná, ako predtým. Po každej fáze refaktorovania je nutné pretestovať doterajší kód a previesť prípadne opravy chýb. Až potom je možné pokračovať ďalšími úpravami.

Upraviť architektúru je možné napr. i z dôvodu výkonnosti aplikácie. Niekedy sa časom javí i výborne navrhnutá architektúra systému ako nevhodná, prípadne príliš zložitá. Návrh je často prevádzaný s ohľadom na prípadné ďalšie zmeny a funkcionalitu, ktorá však nikdy nie je implementovaná ani požadovaná. Použitím metód refaktoringu je možné postupne zložitú architektúru zjednodušiť a nahradiť jednoduchšou, ktorá nebude brať ohľad na príliš mnoho požiadaviek. Tento prístup sa v aplikáciách často i vyskytuje. Pôvodný architekt navrhne aplikáciu príliš komplexne, implementácia bude rozsiahla, no množstvo častí architektúry ostane dlho nevyužitých. Zložito zavedené dedičnosti vedú k neprehľadnosti a komplikovanému volaniu metód, príliš mnoho rozhraní spôsobuje chaos, ktoré z nich je možné použiť. V predošlých kapitolách boli uvedené základné metódy refaktoringu. V skutočnosti je ich veľké množstvo. Dôkladný popis je možné nájsť napr. v prameňoch (2) alebo (1).

Dôvodom na zmenu návrhu aplikácie je často prechod na novú technológiu, ktorá je omnoho výhodnejšia. Moderné technológie umožňujú programátorom koncentrovať sa viac na zákaznícke požiadavky, funkcionalitu a riadenie procesov v programe. Príkladom je rozvoj webových technológii. V porovnaním s pôvodným prístupom známym z jazyka php je dnešný prístup úplne iný. Kým php umožňovalo všetko riadiť na základe prístupu k request, response, session, prípadne ďalším položkám, moderné technológie uprednostňujú prístup podobný desktopovým aplikáciám. Vyskytujú sa v nich udalosti, validátory, hotové komponenty, ktoré programátor iba skladá na stránku. Vývojom prešli aj technológie databázovej vrstvy. Pokiaľ kedysi museli programátori poznať presnú syntax príkazov jazyka SQL, dnes existujú technológie, ktoré komunikujú s rôznymi typmi databáz a programátor musí ovládať len jeden jazyk. Pokiaľ aplikácia komunikovala predtým s databázami napr. MSSQL a ORACLE, musel existovať framework, ktorý umožňoval prevod rozdielnych názvov funkcií a príkazov do dialektu konkrétnej databázy. Ak použije programátor novú technológiu, táto vrstva odpadá a celkový návrh bude jednoduchší.

1.5 Refaktoring a softwarový proces

Existuje niekoľko modelov riadenia vývoja softwarového diela. Existujú modely staršie, ktoré boli používané v časoch, kedy programovacie jazyky boli prevažne štrukturálne a programy neboli natoľko rozsiahle. Samotné aplikácie prevádzali len niekoľko matematických výpočtov, transformácií a výpisov. Časom, ako sa vyvíjali počítače a napredovali technológie, menili sa i programovacie jazyky. Štrukturálne programovanie prešlo v objektovo orientované a aplikácie prevádzali viac, než len výpočty. Počet riadkov zdrojových kódov rástol a projekt nebolo možné riadiť použitím starých metód. Preto vznikali aj nové modely vývoja softwaru. Tieto umožňujú rýchlu správu zmien, včasné odhaľovanie chýb a zlepšili kvalitu produktov. Existuje niekoľko prístupov, ktoré sú v dnešnej dobe rozšírené.

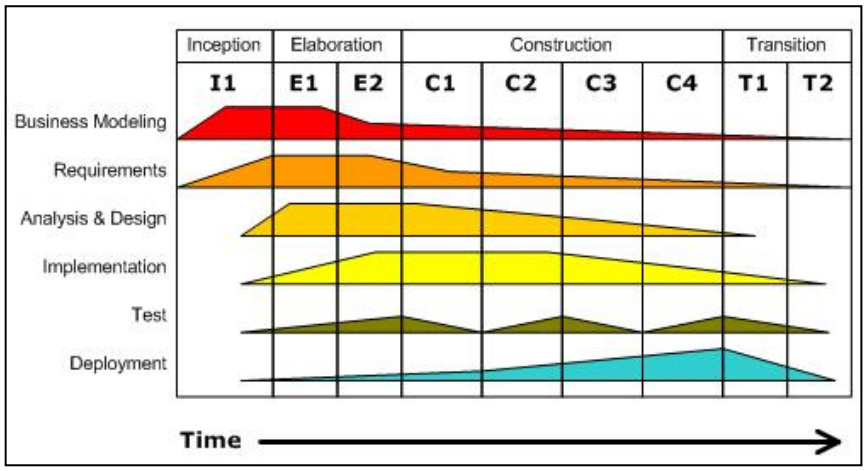


OBR. 3 VODOPÁDOVÝ MODEL

Vodopádový model je najstarším modelom a predstavoval istú mieru ideálneho vývoja softwarového diela, ktorý však nie je v realite a hlavne pri rozsiahlych projektoch možný. Tento prístup bol využívaný ako prvý, no čoskoro sa prišlo na jeho nedostatky. Ako je vidno na Obr. 3, v tomto modeli prebieha fáza analýzy a návrhu (*design*) iba na počiatku realizácie projektu. Ďalej tieto fázy vôbec nevstupujú do diania projektu. Nasleduje už len fáza implementácie (*Implementation*) a testovania (*Verification*). Samozrejme je nemožné poznať všetky požiadavky zákazníka na projekt už pri jeho začiatku. Navyše i fáza implementácie v jednom kroku prináša nevýhodu. Všetky požiadavky sú naraz implementované na základe návrhu a až v ďalšej fáze sú testované, či odpovedajú návrhu. Zároveň samotný návrh nemusí odpovedať skutočným požiadavkám, čo v konečnom dôsledku

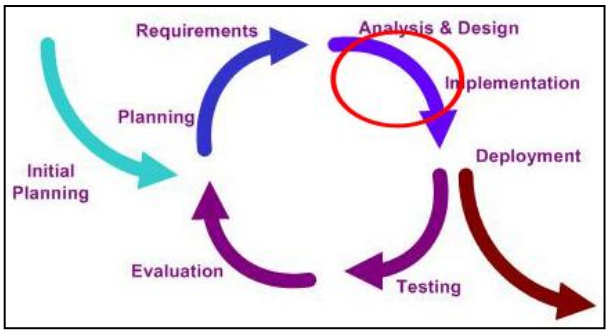
znehodnotí celú prácu vykonanú vo fáze implementácie. V takomto vodopádovom modeli nie je prakticky možné uvažovať o refaktoringu. Keďže všetko, čo je implementované v implementačnej fáze podlieha predchádzajúcemu návrhu. I keď by sa prišlo na chyby v návrhu, výkonnosti a iných častiach, refaktoring by bol veľký na to, aby mohol byť prevedený. Prakticky to je možné považovať za nemožné, pokiaľ by zistená chyba bola v architektúre systému. Na túto chybu by určite nadväzovali ďalšie vrstvy a zmena by musela byť šírená rozsiahlym spôsobom. Vodopádový model je možné preto použiť pri malých projektoch (školské projekty a práce), no nie vo firmách implementujúcich informačné systémy a aplikácie rozsiahleho charakteru.

Naproti tomu iteratívny vývoj poskytuje vysokú mieru pružnosti pri zmenách požiadaviek a na prevedenie refaktoringu. Keďže celý projekt pri iteratívnom prístupe je zložený z niekoľkých iterácii (viď Obr. 4) je možné zmeny previesť rýchlo. Pri vývoji napr. modulu aplikácie nie je nutné implementovať všetku funkcionalitu naraz, ale rozdeliť do niekoľkých iterácií, kde súčasťou každej bude analýza, návrh, implementácia a testy. Takto je napr. pri analýze možné ľahko rozpoznať chyby prevedené v predošlej fáze, nedostatky, prípadne zmeny potrebné na implementáciu nových požiadaviek. Jedna iterácia pri tomto procese trvá približne dva až tri týždne.



OBR. 4 FÁZY ITERATÍVNEHO VÝVOJA SOFTWARU

Možnosť odhalenia chýb a nezrovnalostí návrhu je teda vysoká. To umožňuje okamžité riešenie a preplánovanie projektu. Zahrnúť refaktoring do tohto procesu je veľmi jednoduché. Refaktoring je možné zaradiť ako ďalšiu časť jednej iterácie medzi fázu návrhu a implementácie. Vo fáze návrhu sú spracované nové požiadavky a prepracovaný návrh aplikácie. Samotný refaktoring je možné brať ako prípravnú fázu pred implementáciou. Je možné odstrániť prípadné nezrovnalosti, ktoré by pri samotnej implementácii znamenali problémy. Je však nutné, aby tieto zmeny boli otestované a nenarušili doterajšiu funkcionalitu a nezaviedli nové chyby. Preto je možné za fázu refaktoringu zaviesť fázu testovania. Prípadné problémy, ktoré sú odhalené by vrátili proces znovu do fázy refaktoringu. Takto by bola utvorená malá iterácia. Pokiaľ budú všetky testy úspešné, je možné pokračovať implementáciou novej funkcionality. Celá jedna iterácia by vyzerala podľa Obr. 5. Červenou farbou je vyznačené miesto, kde je možné prevádzať refaktoring. Jedna iterácia by sa podľa závažnosti zmien nemala predĺžiť viac ako o päť dní. Ak by mal refaktoring zabrať väčší časový úsek, je nutné uvažovať o samostatnej iterácii venovanej iba refaktoringu a preplánovať projekt. Tento prístup k refaktoringu umožňuje metóda iteratívneho vývoja softwaru.



OBR. 5 JEDNA ITERÁCIA ITERATÍVNEHO VÝVOJA SW

Okrem predošlého prístupu existuje aj technika tzv. „*extreme programming*“ (ďalej len XP). Táto je špeciálnym prístupom k vývoji. Neexistujú presne definované pravidlá, ktoré hovoria, čo je nutné k prevedeniu takéhoto prístupu. Základom je však programovanie, ktoré je ihneď testované. Účelom je minimalizácia počtu chýb, ktoré sa dostanú do ďalšej fázy projektu. Tento prístup je výhodný i pre refaktoring. Každá implementovaná časť musí podľahnúť vopred pripraveným testom, čo urýchľuje dokonca samotný refaktoring. Testy sú súčasťou tohto projektu. Pokiaľ neexistuje súbor testov pre danú funkcionalitu, nejedná sa potom o metódu XP. Ďalej pri metóde XP je snaha skracovať dobu medzi dodaním funkčných verzií aplikácie zákazníkovi tak, aby mohol testovať správnosť riešenia a kvalitu produktu. Pokiaľ sa vyskytnú chyby, môžu byť ihneď odstraňované. Pri XP je občas spomínané, že programátori pracujú v dvojiciach. Samozrejme to nie je nutnou podmienkou. V neposlednom rade ide však aj o čistotu samotného kódu aplikácie. Pri XP je kladený dôraz i na kvalitu. Kód musí byť naprogramovaný tak, aby spĺňal pravidlá slušného programovania, nezavádzal duplicity a nebol zložitý. Refaktoring je týmto pravidelnou súčasťou celého cyklu vývoja softwaru metódou XP. Refaktorovať je možné kedykoľvek pred implementáciou novej funkcionality, no kód, ktorý budeme refaktorovať musel prejsť všetky testy. Okrem toho po refaktorovaní musí byť spustená sada testov znovu, aby bola overená bezchybnosť kódu.

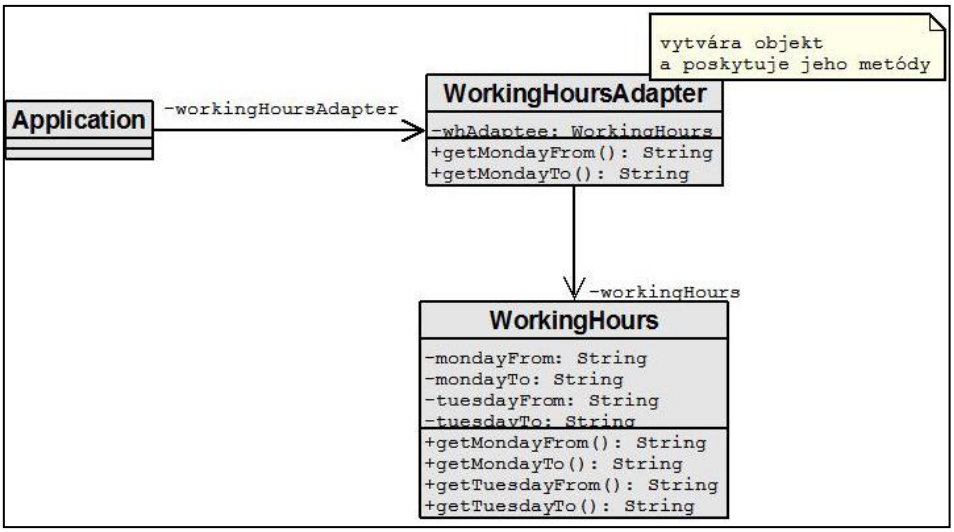
1.6 Refaktoring a návrhové vzory

Návrhové vzory sú bežnou súčasťou programátorskej praxe dnešnej doby. Návrhový vzor je všeobecné riešenie konkrétneho problému, ktoré je možné pri výskyte daného problému vždy aplikovať. Toto riešenie je presne dané a otestované. Slúžia ako zjednodušenie práce pre programátora i architekta systému. Existuje istý počet problémov, pre ktoré existujú už navrhnuté riešenia. Pokiaľ je to možné, návrhové vzory by mali byť zaradené na riešenie daného problému ihneď už pri fáze návrhu. Takto budú programátori ihneď vedieť, že je nutné použiť konkrétny návrhový vzor a architektúra systému bude jednotná. Pokiaľ k tomu nedôjde, môžu programátori implementovať vlastné riešenie, ktoré nie vždy bude dostatočne prehľadné a vhodné.

```
String mondayFrom = (String)request.getAttribute("mondayFrom");
String mondayTo = (String)request.getAttribute("mondayTo");
String tuesdayFrom = (String)request.getAttribute("tuesdayFrom");
String tuesdayTo = (String)request.getAttribute("tuesdayTo");
...
WorkingHours wh = new WorkingHours(
                        mondayFrom, mondayTo, tuesdayFrom, tuesdayTo,
                    ...);
```

ZK. 18 NAČÍTANIE DÁT Z REQUESTU (NEREFAKTOROVANÉ)

Na príklade ZK. 18 je časť kódu, ktorý bol implementovaný programátorom. Keďže sa jedná o servletovú technológiu, kód načíta dáta z requestu a následne ich odošle objektu nazvanom *WorkingHours*. Tento objekt obsahuje údaje o časoch behu služby. Objekt je využívaný pre metódu *countReactionTimes*, ktorá sa stará o počítanie reakčných časov pri incidente. Príklad je pre prehľadnosť skrátený, no v skutočnosti zaberá veľký počet riadkov kódu, pretože takto sú načítané časy od – do pre všetky dni v týždni. Celkom teda kód načítania údajov z requestu zaberie 14 riadkov. Navyše je toto situácia, kedy je iba vytváraná nová služba. Okrem toho je možné službu aj editovať, či duplikovať, a tak sa rovnaká časť kódu bude vyskytovať aj v týchto triedach. Pre riešenie daného problému, je potrebné časť kódu vyňať a prepísať tak, aby bolo možné použiť objekt zo všetkých miest výskytu a nespôsobovalo to duplicity v kóde. Jedným riešením je použitie návrhového vzoru *Adapter*. Návrhový vzor je konštruovaný podľa obrázku Obr. 6. Adaptér teda pôsobí ako konverzný člen, pomocou ktorého je možné, aby údaje boli jednoducho načítané do objektu, kde ich potrebujeme. Samozrejme pre prehľadnosť nie sú zobrazené všetky metódy a vlastnosti, ktoré objekty majú v skutočnosti. Riešenie triedy pomocou adaptéru je na príklade ZK. 19



OBR. 6 TRIEDNY DIAGRAM ADAPTÉRU

Adaptér sa javí v tejto situácii ako výhodné riešenie. Prehľadne adaptuje údaje do objektu, ktorý potrebujeme naplniť, odstraňuje duplicitu načítania v mnohých triedach a takto znižuje celkové riziko chyby v projekte. Pri detailnejšom pohľade je však možné zistiť nedostatok tohto riešenia. Je ním prístup k objektu (a jeho metódam), ktorý adaptér napĺňa. Buď musí adaptér obsahovať metódu, ktorá nám tento objekt sprístupní, alebo musí adaptér definovať všetky metódy, ktoré adaptovaný objekt obsahuje. Druhé riešenie je výhodné iba vtedy, ak chceme sprístupniť iba istý počet metód adaptovaného objektu, prípadne ak dáta musia byť ešte istým spôsobom adaptované pre vstup / výstup (napr. metóda adaptovaného objektu vracia typ *Timestamp*, no pre výstup je potrebný typ *String*. Rovnako zo vstupu prichádzajú údaje v type *String*, no adaptovaný objekt akceptuje iba *Timestamp*). Pokiaľ sa žiadne prevody nekonajú a nepotrebujeme ani limitovať počet prístupných metód, potom je výhodné prvé riešenie, teda vytvoriť metódu, ktorá nám vráti adaptovaný objekt. Treba potom dbať na to, aby programátor nemohol zneužiť takúto hierarchiu iným spôsobom, ako je určená. Takto by mohli vzniknúť problémy a spôsobovali by zbytočné chyby.

K tomuto problému existuje ešte jedno riešenie. V prípade, že je potrebné iba načítať dáta a vyplniť ich do objektu, potom nie je nutné použiť návrhový vzor *Adapter*. Riešením je napr. predanie objektu, odkiaľ sa dáta načítajú ako argument konštruktoru. Dáta sú potom v konštruktore vybrané z objektu a správne nastavené do vlastností nového objektu. Samozrejmosťou je ošetrenie

prípadných problémov s načítaním údajov. Pokiaľ nie je možné načítať parameter, je potrebné to oznámiť výnimkou.

```
public class WorkingHoursAdapter {
  private WorkingHours whAdaptee;

  public WorkingHoursAdapter(HttpServletRequest request) {
      this.whAdaptee = readFromRequest(request);
  }

  private WorkingHours readFromRequest(HttpServletRequest request) {
      WorkingHours wh = new WorkingHours();
      wh.setMondayFrom((String) request.getAttribute("mondayFrom"));
      wh.setMondayTo((String) request.getAttribute("mondayTo"));
      wh.setTuesdayFrom((String) request.getAttribute("tuesdayFrom"));
      wh.setTuesdayTo((String) request.getAttribute("tuesdayTo"));
      ...
      return wh;
  }

  public WorkingHours getWorkingHours() {
      return this.whAdaptee;
  }
}
```

ZK. 19 TRIEDA ADAPTÉRU (POUŽITÝ PRVÝ TYP PRÍSTUPU)

Pre toto riešenie existuje ešte jedná možná modifikácia, ktorá ale má nedostatky. Často je pridaná do objektu, ktorý sa má načítať iba nová metóda, ktorá sa o naplnenie postará. Toto má však nevýhodu v tom, že programátor, ktorý nepozná dokonale triedu s ktorou pracuje nezavolá metódu na načítanie a údaje začne načítať spôsobom uvedeným na príklade ZK. 18. Práve kvôli tomuto musia byť vlastné triedy dostatočne okomentované, prípadne poskytovať v dokumentácii jednoduchý príklad, ktorý je ľahko pochopiteľný a jasný. V opačnom prípade je pohodlnejšie použiť návrhový vzor.

Ďalším príkladom refaktoringu a návrhových vzorov je možné použitie továrnych (*Factory*) návrhových vzorov. Pokiaľ sa v aplikácii nachádza napr. logická vrstva starajúca sa o spracovanie údajov pre databázovú vrstvu, potom určite nebude existovať iba jedna trieda logickej vrstvy, ale mnoho iných. Aby sa dosiahlo sprehľadnenie prístupu k triedam, je zavedený návrhový vzor *Factory*. Toto riešenie v konečnom dôsledku nielen sprehľadňuje prístup k objektom logickej vrstvy, ale umožňuje vyššiu mieru flexibility. Táto sa prejaví napr. pri nutnosti náhrady starej vrstvy novou. Jednoduchá zmena prebehne iba v továrnej triede, nie však zdĺhavým hľadaním v kóde všetkých tried. Možná implementácia je na príklade ZK. 20, triedny diagram na obrázku Obr. 7.

. Volanie konkrétnej triedy na logickej vrstve je veľmi jednoduché, ako vidno na príklade. Navyše má vzor výhodu, že je ho možné prepracovať tak, aby boli jednotlivé triedy ako *Singleton* a neboli vytvárané nové objekty.

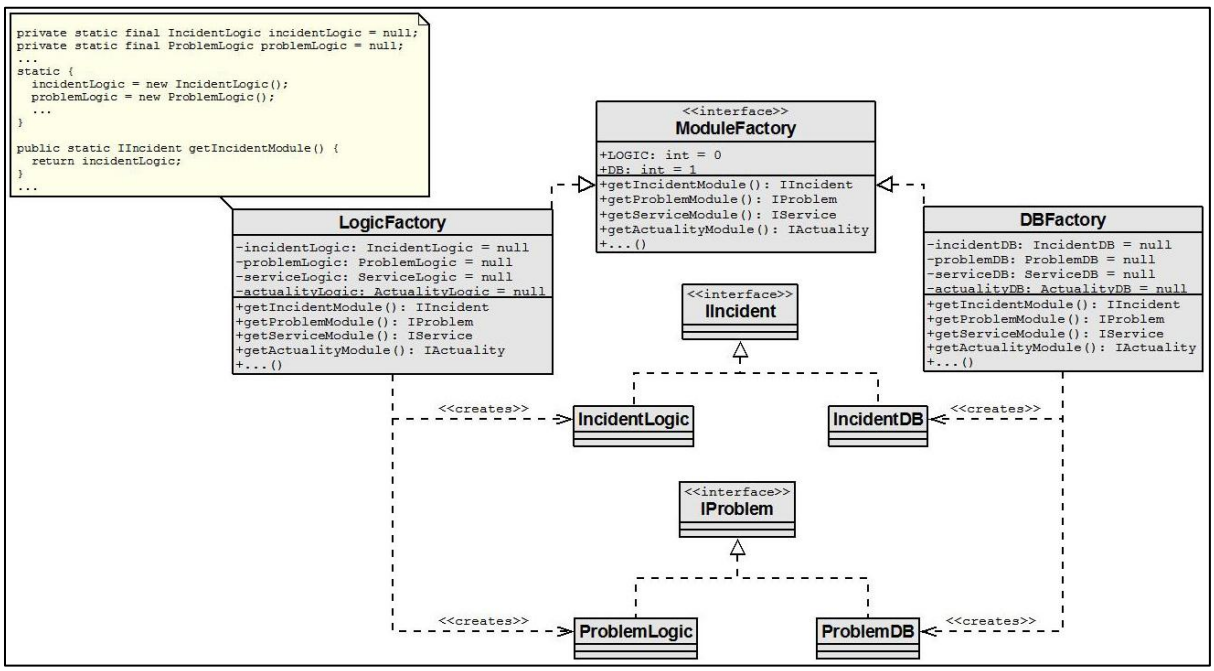
Návrhové vzory nie sú len o prinášaní štruktúry a riešenia do problému. Pôvodné riešenie je niekedy rovnako prehľadné ako riešenie s použitím návrhových vzorov. Existujú však návrhové vzory, ktoré zvyšujú i výkon celej aplikácie. Takýmto je i napr. návrhový vzor *Singleton*, prípadne tzv. *Object Pool*. Oba sprostredkujú prístup k obmedzenému zdroju, inicializujú svoje interné objekty buď pri svojom vytvorení alebo pri určitej udalosti alebo volaní. Použitie object poolu je výhodné napr. pri pripojeniach k databáze. Object pool si udržuje počet vytvorených pripojení v tzv. *poole*. Každé

pripojenie je potom viazané napr. na určitého užívateľa. Pri jeho prihlásení, je vytvorené nové pripojenie alebo vybrané už vytvorené, ale neaktívne pripojenie z poolu.

```
public static class Application {
  public ModuleFactory getLayer(int layerID) {
        switch(layerID) {
        case ModuleFactory.LOGIC:
          return new LogicFactory();
        case ModuleFactory.DB:
          return new DBFactory();
        }
        throw new IllegalArgumentException("Illegal layer ID");
  }
}
...
ModuleFactory factory = Application.getLayer(ModuleFactory.LOGIC);
factory.getIncidentLogic().insert(incidentDTO);
```

ZK. 20 VOLANIE TRIED PRI POUŽITÍ ABSTRACT FACTORY

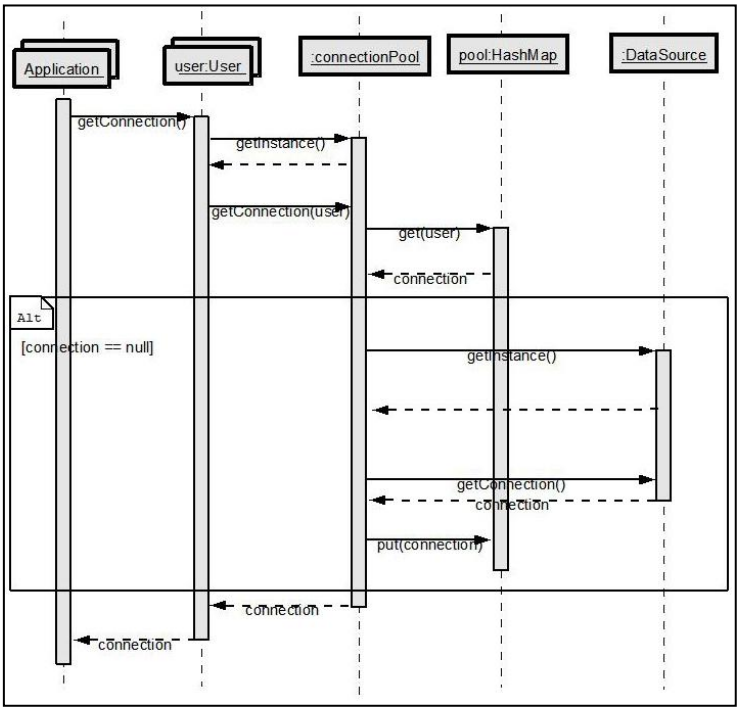
Pri odhlásení je toto pripojenie odstránené alebo vložené do poolu ako neaktívne (napr. z dôvodu nesprávneho odhlásenia užívateľa). Výhoda takéhoto riešenia spočíva v tom, že vytvorenie pripojenia na databázu trvá určitú dobu. Toto oneskorenie môže spôsobovať problém s výkonom na stránkach alebo v aplikácii. Ak napr. vytvorenie pripojenia bude trvať 0,5s a samotný SQL dotaz zaberie 0,05s je to veľmi neefektívny spôsob práce. Object pool musí udržiavať svoj pool v rýchlej štruktúre (napr. hashovacia mapa alebo strom), kde vyhľadanie a vrátenie objektu zaberie veľmi krátku dobu. Pokiaľ bude databázové pripojenie vyžadované bude vrátené a označené ako zamknuté (angl. locked). Ak bude ukončené jeho používanie, bude vrátené do zoznamu voľných pripojení. Takto je možné pomocou použitia návrhového vzoru zároveň zvýšiť efektivitu aplikácie. Sekvenčný diagram interakcie takéhoto systému je na Obr. 8.



OBR. 7 NÁVRHOVÝ VZOR ABSTRACT FACTORY V PRAXI (TRIEDNY DIAGRAM VZORU)

Scenár obsahuje úspešné získanie pripojenia na databázu a jeden alternatívny scenár. Alternatívny scenár nastane vtedy, keď z poolu pripojení je vrátená hodnota *null*, čo znamená, že pre daného

užívateľa neexistuje žiadne pripojenie. V tomto prípade je zaslaný dotaz na správcu pripojení o vytvorenie nového pripojenia. Toto je následne zaslané potom na najvyššiu vrstvu a je možné pracovať s pripojením. Pripojenie musí byť po dokončení práce vrátene naspäť do poolu, prípadne uzavrené, pokiaľ sa užívateľ odhlásil. Zároveň je potrebné dbať na správne spravovanie pripojení, a to hlavne ak došlo k nesprávnemu uzavretiu aplikácie a pripojenie ostalo v poole. V tomto prípade musí existovať doba, po ktorej sa takéto pripojenia uzavrú. O toto sa môže starať napríklad osobitné vlákno alebo iný princíp.



OBR. 8 SEKVENČNÝ DIAGRAM ZÍSKANIA PRIPOJENIA (CONNECTION)

Pri refaktoringu je vždy potrebné prehodnotiť, či nie je možné použiť návrhový vzor, ktorý ponúka už hotové riešenie. Vzory sú veľmi dobré riešenia, ktoré zjednodušujú prácu programátora. V istých situáciách je nutné však dávať pozor, pretože niektoré návrhové vzory trpia nedostatkami. Spôsobujú potom zbytočné duplicity v kóde a prinesú opačný výsledok, ako je od refaktoringu očakávané. Pri refaktoringu by sa mali duplicity v čo najväčšej miere odstraňovať a nie zavádzať. Rovnako veľké množstvo nových (často zbytočných) rozhraní, ktoré návrhové vzory obsahujú spôsobujú neprehľadnosť. Vysoká miera abstrakcie nie je vždy vhodná pre programátorov. Ak má refaktoring priniesť spolu s návrhovými vzormi výhodu, musí byť prevedený dôkladne. Návrhové vzory nie je vždy nutné použiť tak, ako sú definované. Po úprave a prispôsobeniu pre konkrétnu situáciu budú lepšie spĺňať požiadavky na ich funkciu a prehľadnosť.

Niekedy je pri refaktoringu zložité rozhodnutie, ktorý návrhový vzor spĺňa funkciu najlepšie. V takomto prípade je možné použiť i viac návrhových vzorov naraz. Kombinácia návrhových vzorov alebo ich modifikácia prináša vyššiu mieru prehľadnosti do výsledného kódu, ako použitie pôvodných návrhových vzorov. Pre refaktoring do návrhových vzorov je však aj potrebné dodržať istý postup.

Orientačné body refaktoringu na návrhové vzory:

- Pôvodné rozhrania, triedy a metódy nemeniť
- Najprv vytvoriť návrhový vzor aplikovaný na konkrétnu situáciu (Vytvoriť novú triedu *Singleton*, vytvoriť novú továreň, adaptér)

- Snažiť sa čo najmenej zasahovať do pôvodného kódu
- Otestovať funkčnosť návrhového vzoru pomocou vzorkovej triedy (Vybrať jednu z tried využívajúcich pôvodné riešenie a previesť ho na nové)
- Postupne prevádzať na nové riešenie všetky ostatné triedy (Pri použití *Singleton* tried to nie je možné, potom treba previesť najprv analýzu, kde je treba spraviť zmenu a následne previesť všetky úpravy a otestovať). Každú pôvodnú triedu premenovať s určitým prefixom / postfixom, aby bolo možné sa kedykoľvek v prípade problému vrátiť k riešeniu. Nové triedy budú pomenované ako pôvodné (Pri správe projektu pomocou CVS alebo SVN je možné pre pôvodné súbory zaviesť *TAG* – značku, ktorá bude označovať miesto, kedy súbory boli pred zmenou a takto nie je nutné používať zmeny názvov)
- Pôvodné triedy označiť ako *deprecated*, prípadne upozorniť v dokumentácii, že budú zmazané (V prípade použitia CVS / SVN tagov nie je nutné, pretože tag označuje miesto od ktorého sa súbory menili a je možné kedykoľvek sa k tejto verzii vrátiť. Do súboru je potom iba potrebné zaviesť komentár v hlavičke, že bol prevedený refaktoring)

Pri refaktoringu je použitie návrhových vzorov, rovnako ako pri návrhu aplikácie, výhodný. Je však nutné zvážiť v akej miere je výhodné použitie vzoru a koľko času bude venovaných prevodu a aký bude prínos. Niekedy je pri implementovanom riešení neefektívne robiť veľké zmeny, ktoré by spomaľovali vývoj novej súčasti. V konečnom dôsledku je zavedenie návrhových vzorov typu Object pool veľmi náročné, pretože je potrebné naraz refaktorovať veľké množstvo tried. V tomto prípade nie je možné prevádzať refaktoring postupne, ako tomu je v prípade zavedenia adaptéru. Pre zavedenie tak rozsiahlych refaktoringov je potrebné pozastaviť dočasne vývoj a previesť aspoň základný refaktoring a uviesť ho do testovania. Po úspešnom otestovaní je potrebné nasadenie a potom už len postupné upravovanie do konečného štádia.

1.7 Výhody – nevýhody refaktorovania

V súčasnosti existuje mnoho metodik pri vývoji softwaru. Každá z nich disponuje mierou efektivity a každá má svoje nedostatky. V praxi sú tieto metodiky často krát modifikované tak, aby dosahovali čo najnižšie náklady pri vývoji softwaru a maximálnu produktivitu. Znamená to, že pri vývoji sú zanedbávané procesy, ktoré by predlžovali vývoj a zvyšovali jeho cenu. Pri malých projektoch, ktoré sú iba vyvinuté a predané zákazníkovi je táto metodika výhodná. Veľké projekty, ktoré majú pokračovať a rozširovať sa postupom času nesmú mať takýto prístup k vývoju. V takýchto projektoch sa potom objavuje refaktoring. Jeho výhodnosť je možné hodnotiť vzhľadom na jeho prínos do projektu. Výhodou dobrého refaktoringu je vysoká kvalita výstupného kódu, ľahké hľadanie miest výskytu chýb a často i ľahšia optimalizácia výkonu aplikácie.

Výhody – nevýhody refaktoringu

+ vyššia prehľadnosť kódu	- Náklady spojené s prevedením refaktoringu (čas, ktorý programátor strávi refaktoringom nie produktívny čas programátora – neimplementuje novú funkcionalitu)
+ rýchle odstránenie chýb	
+ optimalizácia výkonu	
+ odhalenie chýb a možných nedostatkov	
+ rýchlejšia implementácia novej funkcionality	- Časové posuny v termínoch
+ zníženie nákladov na opravy	- Nie vždy prináša očakávaný výsledok

Hlavnou nevýhodou refaktoringu je v konečnom dôsledku čas, ktorý programátori musia venovať organizácii kódu a neprodukujú žiadnu hodnotu. Z toho plynie, že refaktoring je platený z interných zdrojov a pre firmu predstavuje investíciu, ktorá sa nie vždy musí vrátiť. Investície do refaktoringu nie sú manažéri ochotní dávať a preto je refaktoring potláčaný ako nevýhodný. Ak však programátori

pracujú často pod tlakom termínov, je refaktoring výhodne prevádzať v čase, kedy vývoj nie je pod tlakom termínov. Čas strávený takýmto refaktoringom nespôsobí veľký výpadok vo vývoji a strávený čas bude vrátený pri implementácii nových funkcií.

1.8 Správny refaktoring

Rovnako ako nie je možné previesť ideálny návrh, nie je možné robiť ideálny refaktoring, keďže refaktoring je vo svojej podstate zakaždým i nový návrh architektúry aplikácie. Možnosťou pri prevedení refaktoringu je dôkladné zváženie, ktoré kroky je potrebné previesť, a ktoré je možné presunúť do ďalšej fázy refaktorovania. Ďalšou dôležitým faktom je hĺbka refaktorovania. Refaktorovanie by malo byť sústredené vždy na jednu konkrétnu triedu, nad ktorou sú prevádzané úpravy. Pokiaľ je prevádzaný iba rozklad kódu na menšie časti, je toto možné vždy. Refaktorovanie viac tried naraz je potrebné analyzovať. Refaktoring nemôže byť prevedený hlbšie ako o stupeň v hierarchii volaní. Znamená to, že ak je refaktoring prevádzaný na triede volajúcu metódy inej triedy, je možné refaktorovať aj volanú metódu, no nie už jej ďalšie volané metódy. Pri nedodržaní tohto pravidla nastane situácia, kedy ostane mnoho rozpracovaného kódu, ale refaktoring bude zanorený hlbšie do hierarchie a nebude nakoniec úspešný. Pri výskyte nezrovnalostí v hlbšej hierarchii je možné buď zalogovať nedostatky a nechať ich otvorené v dokumente, ktorý je používaný pre správu, alebo nerefaktorovať vôbec. Po prevedení refaktoringu menšej časti je dobré previesť uloženie na CVS / SVN pod vhodným popisom, prípadne zaviesť tag.

Pokiaľ je na kompletný refaktoring triedy nedostatok času, postačuje, keď sú refaktorované iba niektoré časti. Ostatné môžu byť prevedené iným programátorom alebo pri ďalšej iterácii. I takýto neúplný spôsob výrazne prispieva k udržaniu kvality kódu. Uprednostnené by mali byť hlavne časti, ktoré sa často opakujú v kóde triedy, číselné hodnoty, ktoré by mali byť ako konštanty, prípadne dlhé úseky kódu, ktoré je možné vyňať do samostatnej metódy. Ak bude refaktoring prevádzaný na viacerých triedach, potom je potrebné dbať o dodržanie pôvodnej funkčnosti kódu. Za každou takouto zmenou by mali byť prevedené testy podľa testovacích scenárov. Pokiaľ existujú automatické testy, potom je nutné ich spúšťať vždy po prevedení refaktoringu nad triedou a overiť správnosť výsledkov. Ak bola refaktoringom zavedená chyba, bude včas odhalená a odstránená. Ak budú testy pustené príliš neskoro, nájdenie a odstránenie chyby bude nákladnejšie.

Doporučenia refaktoringu:

- Refaktorovať iba konkrétnu triedu, nad ktorou majú byť prevedené zmeny
- Najprv previesť jednoduché refaktorovanie (prevedenie hodnoty na konštantu, vyňatie opakujúceho sa kódu do vlastnej metódy, zjednodušenie podmienok, ...)
- Refaktoring previesť maximálne do hĺbky prvého stupňa (refaktorovať maximálne metódy volané z konkrétnej triedy, nie už však metódy volané hlbšie)
- Ak nie je možné refaktorovať kompletnú triedu, potom previesť logovanie nedostatkov, ktoré je potrebné odstrániť, v prípade malých zmien nie je refaktoring nutný (bude prevedený inokedy)
- Čas strávený refaktoringom by nemal byť dlhší ako polovičný čas strávený vývojom nad konkrétnou triedou (Ak je na vývoj napr. 5 hodín, potom refaktoring by nemal presiahnuť 2.5h, dlhší čas už je rozsiahly refaktoring a predlžuje zbytočne čas vývoja)
- Pokiaľ je niektorá časť systému refaktorovaná, nie je možné prevádzať na nej paralelný vývoj (ostatní programátori nesmú zasahovať do kódu)
- Refaktoring prevádzať len nad triedami, ktoré úspešne prešli testami (nie je vylúčené, že pri refaktorovaní budú odhalené ďalšie chyby, ktoré testy nepreukázali. Potom je nutné zapísať chybu a odstrániť ju pred refaktoringom)

- V prípade veľkých refaktorovaní previesť dôkladnú analýzu, ktoré časti systému budú zasiahnuté (nebude s nimi možné pracovať)
- Prevádzať pravidelne po každej zmene kontrolu funkčnosti (nie je nutné pri malých zmenách, pri zmenách cyklov, presune metód medzi triedami, mazanie starých častí kódu, optimalizácia výkonu a pod. je nutné)

1.9 Refaktoring a customizácia

Slovo customizácia sa pomerne často vyskytuje pri obchode so softwarom. Vyskytuje sa hlavne pri softwarových produktoch, ktoré majú veľký rozsah pôsobnosti. Produkty ako helpdesk, skladový software a iné nemôžu byť výrobcom vytvorené pre potreby každého zákazníka. Práve preto sú na začiatku dohodnuté prípadne zmeny a úpravy, ktoré prevedie výrobca pre zákazníka. Toto sa nazýva customizácia (angl. *customization*). Rozsah prevedených úprav a zmien je zakotvený v kúpnej zmluve medzi zákazníkom a výrobcom. Na výrobcovi ostáva potom, ako prevedie správu zmien pre jednotlivých zákazníkov a či zmeny budú prevedené i do pôvodného softwaru alebo iba vyhradené ako customizácia.

Customizácia, ktorá ostáva vyhradená iba konkrétnym zákazníkom nespôsobuje problémy v softwari. Je však potrebné viesť evidenciu o zmenách, ktoré boli prevedené. V prípade predania vyššej verzie zákazníkovi totiž musí jeho verzia obsahovať všetky zmeny, ktoré boli vykonané. Je možné ukladať tieto zmeny na CVS / SVN pod konkrétnymi tagmi a takto mať prehľad o zmenách a možnosť vrátiť sa k pôvodnej verzii. V prípade implementácie customizácie týmto spôsobom nie je nutný refaktoring. Keďže zmena býva zavedená iba pre konkrétneho zákazníka, úpravy kódu spôsobené refaktoringom by nemali význam a boli by zbytočné. Navyše tento spôsob by smeroval k príliš otvorenej architektúre, kde by ostávali potom nevyužité časti a robili systém len viac komplexným. Toto rozhodne nie je dobrý postup. Pokiaľ budú zmeny správne navrhnuté a implementované, nie je nutnosť meniť ostatnú architektúru systému.

Druhý typ customizácie, ktorý bude zavedený i do softwaru ako štandardná súčasť by mal podliehať správe, ako bežný vývoj nad softwarom. Pokiaľ teda bude zavedená súčasť do systému, musí byť zavedená tak, aby to nebolo v rozpore s architektúrou systému a prípadné nezrovnalosti je potrebné riešiť novým návrhom a prevedením refaktoringu. Pri refaktorovaní je treba dbať na zásady správneho refaktoringu a udržiavať refaktoring pod kontrolou. V opačnom prípade povedie k neúspechu a programovanie bude nutné opakovať, prípadne vrátiť zmeny do pôvodného stavu a začať refaktoring znovu. Vynaložené finančné prostriedky je možné dodržaním správnych zásad minimalizovať a zvýšiť efektivitu refaktorovania.

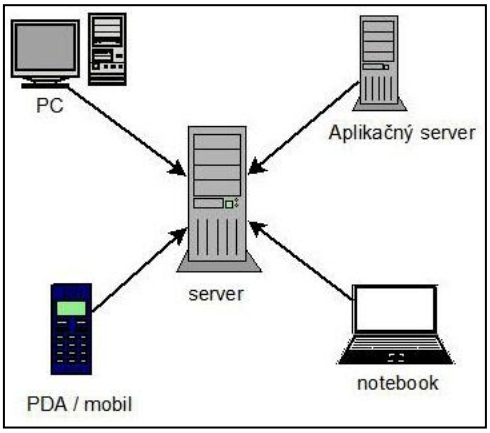
2. Zmena architektúry systému

2.1 Architektúra systému

Významným prvkom všetkých softwarových systémov je ich samotná architektúra. Architektúra je v podstate základ, na ktorom systém bude v budúcnosti stáť. Rovnako ako v stavebníctve je nutné i pri vývoji softwaru mať pevný základ, ktorý udrží v konečnom dôsledku celý systém bez problémov. Architektúra systému je vyvíjaná predovšetkým v počiatočných fázach softwarového procesu. V tomto momente sú definované základné požiadavky na systém a je stanovený približný náčrt jeho funkcií. Človek, prípadne skupina ľudí, ktorí vyvíjajú základ systému sú nazvaní architektmi systému. Pri jej návrhu musia brať v úvahu mnoho faktov a počítať v istej miere i do budúcnosti. Architektúra systému musí byť dostatočne pružná, no i v tomto prípade je potrebné dbať na spravovateľnosť systému a jeho architektúry. Pokiaľ architekt navrhne príliš komplexnú architektúru z dôvodu možného rozšírenia systému v budúcnosti, je možné, že táto výhoda bude využitá, ale vo väčšine prípadov spôsobuje iba dlhodobé problémy a dodatočná funkcionalita nie je nakoniec implementovaná. Takáto architektúra je nevhodná. Pri návrhu je potrebné vedieť požiadavky zákazníka, prípadne jeho budúce potreby a zhodnotiť do akej miery bude architektúra pripravená na takéto zmeny. Samozrejme diskusia ohľadne požiadaviek zákazníka odhalí, ktoré požiadavky zákazník uplatní, ktoré naopak nie. Niekedy je vhodné predložiť iné riešenie, ktoré zákazníkovi nahradí jeho požiadavky, ale pri návrhu výrazne prispeje k zjednodušeniu architektúry systému.

V súčasnosti existuje niekoľko typov architektúr, ktoré sa často využívajú v rôznych situáciách. Na využití niektorej záleží len od požiadaviek, ktoré na systém budú kladené. Existujú aplikácie, ktoré využívajú architektúry postavené hlavne na databáze. Databáza v tomto prípade vykonáva všetky operácie pomocou uložených procedúr a funkcií. Veľkou nevýhodou je vysoká miera viazanosti na konkrétny typ databázy.

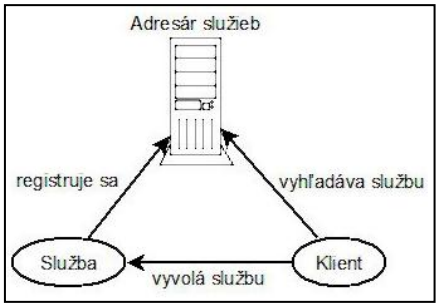
Iným typom architektúry je klient – server. Klient – prevažne klientska aplikácia vo väčšine prípadov len zobrazuje výsledok, dáta zo serveru. Server je hlavným prvkom, na ktorý sú posielané požiadavky o službu. Tento odpovedá potom klientovi a zašle mu výsledok operácie. V tomto prípade je situácia ešte rozdelená na tenkých a tučných klientov. Rozdiel medzi nimi je v miere závislosti na serveri. Kým tenký klient sa spolieha prevažne na činnosť serveru, dáta nespracuje, iba ich zobrazuje, tučný klient má časť funkcionality prenesenú k sebe. Ich použitie závisí iba na situácii. Typickými klient – server aplikáciami sú napr. ftp, cvs, svn, mail a iné.



OBR. 9 ARCHITEKTÚRA KLIENT - SERVER

Na obrázku Obr. 9 je možné vidieť architektúru klient – server. Existuje centrálny server, ktorý poskytuje službu. Jeho službu môžu využívať užívatelia z bežného PC, notebooku alebo mobilného telefónu. Okrem toho môže využívať služby serveru aj napr. aplikačný server, prípadne iné typy serverov, ktoré potrebujú prístup k danej službe. Na obrázku je možné si predstaviť, že server poskytuje napr. emailové služby a každé zo zariadení ma v sebe tenkého klienta. Aplikačný server nemusí obsahovať klienta, ale pozná protokol, ktorým komunikuje server, a preto môže využívať jeho služby.

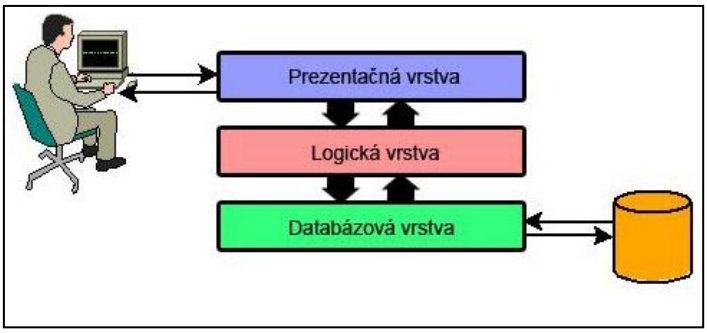
Ďalším typom architektúry, v dnešnej dobe veľmi populárnej, je tzv. servisne orientovaná architektúra (angl. *service oriented architecture*). Celá táto architektúra je postavená na vzájomne komunikujúcich službách. Každá služba je nezávislá od inej služby a všetky služby sú uchovávané v centrálnom zozname. Odtiaľ je možné získať informácie o službe a jej rozhraní a následne komunikovať so službou na základe SOAP protokolu. V tejto architektúre sú všetky (i databázové záležitosti) implementované ako služba. Problémom môže byť rýchlosť tejto architektúry, keďže komunikácia protokolom SOAP sú vlastne XML dokumenty a obsahujú veľké množstvo popisných dát. Okrem toho je nutné službu najprv vyhľadať v centrálnom adresári služieb, získať rozhranie a zaslať potom až správu službe. Do istej miery je výhodné použitie webových služieb, pretože oddelenie webových služieb znamená ich ľahkú zmenu. Pokiaľ sa nemení rozhranie služby, je možné ju kedykoľvek a ľahko modifikovať a zameniť za existujúcu službu. Okrem tohto prináša SOA výhodu opakovaného využitia už existujúcich služieb pre vývoj nových systémov. Hlavne pokiaľ sa firma zaoberá vývojom niekoľkých druhov softwaru, môže mať vytvorených niekoľko služieb a tieto sú súčasťou viacerých aplikácii. Teda šetrí čas programátorov pre novú implementáciu funkčností a programov. Toto však platí iba vtedy, ak sú produkty firmy podobné (napr. firma zaoberajúca sa podnikovými informačnými systémami). Obrázok Obr. 10 znázorňuje princíp SOA (*Service Oriented Architecture*). Klient je reprezentovaný aplikáciou, ktorá potrebuje prístup k službe. Adresár služieb je server, ktorý obsahuje popisy služieb a ich rozhraní. Popisy služieb sú uložené v jazyku WSDL (*Web Service Description Language*). Keď klient dostane od centrálneho adresára popis služby, je schopný vyvolať službu. Pritom použije protokol SOAP. Služby môžu bežať na rôznych strojoch v celej sieti, no každá musí byť zaregistrovaná so svojím popisom v centrálnom adresári, aby ju bolo možné vyhľadať a využiť.



OBR. 10 ARCHITEKTÚRA SOA

Veľmi častým typom architektúry je i tzv. trojvrstvová architektúra (angl. *3-layer architecture*). Jej podstatou je oddelenie prezentačnej vrstvy, databázovej a logickej vrstvy navzájom od seba. Databázová vrstva sa stará iba o komunikáciu s databázou, získava dáta a spracúva ich do dátových objektov. Tieto sú zaslané vyššej logickej vrstve. Tá môže buď modifikovať údaje, ktoré je potrebné upraviť alebo ich len predá prezentačnej vrstve. V opačnom prípade, ak z prezentačnej vrstvy majú byť dáta uložené do databázy, zašle prezentačná vrstva logickej svoje údaje, tá ich predspracuje a pošle databázovej vrstve, ktorá sa postará o ich uloženie. Tento typ je využívaný veľmi často v podnikových aplikáciách a celkovo informačných systémoch. Model takejto architektúry je naObr.

11. Užívateľ prichádza do styku len s najvyššou vrstvou architektúry. Toto môže byť buď rozhranie webovej aplikácie, alebo desktopová aplikácia. V prípade webových aplikácií v java prostredí je možné použiť technológie ako JSF, Struts, ICEFaces alebo mnoho iných. V prípade .NET technológie je to ASP .NET. Pod touto vrstvou sú dve vrstvy, ktoré sa starajú o korektné spracovanie dát. Jedna je zvaná logická vrstva, niekedy označená ako „*Byznys vrstva*“ (angl. *Business layer* alebo *Business logic*). Táto sa stará o správu objektov, ich spracovanie na základe byznys požiadaviek, stará sa o spravné riadenie prechodov medzi stavmi objektov a pod. Logická vrstva je vo svojej podstate framework rozhraní, tried a metód, ktoré sú volané z prezentačnej vrstvy. Pod touto je už databázová vrstva, ktorá komunikuje s databázou. Ako technológiu tejto vrstvy je možné použiť pre java aplikácie napr. Hibernate alebo JPA framework. Pre jednoduchšie aplikacie s jednou databázou toto nie je potrebné a je možné vytvoriť databázovú vrstu postavenú len na základných prvkoch (jazyk java poskytuje tieto prvky pod názvom JDBC – Java DataBase Connectivity) , ktoré poskytuje jazyk, v ktorom implementujeme informačný systém (pre .NET technológiu je použiteľný najnovší jazyk Linq alebo staršia technológia ADO .NET).



OBR. 11 TROJVRSTVÁ ARCHITEKTÚRA

Väčšina zákaznícky orientovaného software je spojená v istej miere s databázou. Je preto nutné počítať s komunikáciou s touto databázou a v architektúre sa prejaví táto skutočnosť existenciou iných vrstiev architektúry ako by tomu bolo napr. pri grafických systémoch, ktoré nevyžadujú databázu. Spoločnou časťou každého systému je jeho prezentačná vrstva. Tá existuje ako pri grafických programoch, tak pri podnikových informačných systémoch, helpdesku alebo iných aplikáciách. Niekedy má systém dokonca niekoľko prezentačných vrstiev (desktop i web rozhranie) pomocou ktorých je možné komunikovať so systémom. Prezentačná vrstva je do istej miery ešte súčasťou architektúry. Je nutné mať definované rozhranie pomocou ktorého sa bude prezentačná vrstva schopná dorozumievať s jadrom systému. Vizualizácia výsledných dát je potom však úplne oddelená od jadra systému. Samotné jadro systému využívajúceho databázu je postavené na komunikácii s databázou a spracovaní údajov pre uloženie do databázy. Samozrejme v dnešnej dobe je pri využití moderných technológii v architektúrach systému bežné postaviť systémy lepším spôsobom ako tomu bolo v minulosti. Hlavným problémom databázových systémov bola podpora viacerých databáz. Keďže zákazníci majú svoje zakúpené licencie na databázy, nie je z ich strany ochota meniť toto riešenie. Výrobcom softwaru ostáva potom len rozšíriť podporu rôznych typov databáz. V minulosti toto bolo riešené zložitými vlastnými konštrukciami, ktoré nie sú veľmi flexibilné. Vo väčšine prípadov ide iba o podporu najrozšírenejších databáz, a to ORACLE a MSSQL. Ostatné sú v dôsledku zložitosti zmien v architektúre zavrhnuté. Práve tento problém je možné riešiť zmenou v architektúre systému a prechodom niektorých častí na modernejšie technológie.

2.2 Refaktoring a zmena architektúry existujúceho systému

Pri návrhu architektúry systému je od počiatku potrebné dbať o kvalitu návrhu. V každej iterácii softwarového procesu musí byť architektúra zhodnotená a zmeny usmernené tak, aby viedli

k prispôsobeniu architektúry aktuálnym požiadavkám analýzy. Každá zmena môže výrazne ovplyvniť výslednú kvalitu systému, i jeho výkonnosť. Z pozície architekta vyplýva vysoká miera zodpovednosti. Niekedy však ani architekt nedokáže predvídať chyby a problémy, ktoré môžu spôsobiť požiadavky zákazníkov. V niektorých prípadoch môže aj dobrý návrh narušiť veľký časový tlak, pod ktorým pracujú programátori. Implementácia celého systému potom prebieha mimo návrh a súčasti nie sú implementované kvalitne. Práca pod tlakom je výsledkom zlého projektového riadenia a plánovania. Dôvodov pre prechod na novú architektúru systému býva niekoľko.

Dôvody zmeny architektúry systému:

- *Súčasná architektúra je príliš komplikovaná*
- *Architektúra systému je zastaraná*
- *Systém je neudržiavateľný v súčasnej podobe*
- *Rozšírenie systému o súčasti, ktoré by pôvodná architektúra nezvládla (v prípade, že refaktoring nie je možný alebo príliš nákladný)*

Dôvody na zmenu architektúry sú samozrejme iba v krajných prípadoch, kedy už neexistuje žiadne iné východisko. Znamená to, že nie je možnosť previesť refaktoring architektúry. Pod pojmom refaktoring architektúry je možné si predstaviť prevádzanie refaktoringu nad triedami architektúry za účelom zvýšenia kvality kódu a optimalizácie riešenia. Príkladom je zjednodušovanie príliš komplexnej architektúry, ktorá sa časom ukázala ako nevyužitá. Refaktoring je rovnako možné použiť pri optimalizácii systému. Neprehľadné časti po úpravách bývajú ľahšie čitateľné a je z nich možné vyčítať úzke miesta pre výkon. I keď všetky tieto možnosti existujú, niekedy je refaktoring starého systému finančne náročnejší ako implementácia nového. V tomto prípade je potom na dohode, v akej miere prebehne implementácia od začiatku. Niektoré časti systému môžu ostať funkčné a byť prebrané do nového.

2.3 Systém ServiceBase

Systém ServiceBase je produkt firmy Crux Information Technology s.r.o. Zameraný je predovšetkým na správu firemnej infraštruktúry a služieb informačných technológií (IT). Okrem toho poskytuje aj spôsob riadenia zmien v tejto infraštruktúre. Je založený na doporučení ITIL. ITIL (angl. *Information Technology Infrastructure Library*) je súbor riešení a doporučení (tzv. *Best practices*), ktoré sú časom preverené pri správe IT služieb. Jeho účelom je prínos pri správe IT zdrojov akými sú siete, počítače, aplikácie a iné zdroje, ktoré firmy potrebujú pre správny beh a fungovanie. Výsledkom je zvýšenie kvality poskytovaných služieb, minimalizácia rizík spojených s IT a finančná efektívnosť pri riešení problémov. ITIL je procesne založený. Obsahuje okrem iného aj doporučenia pre implementáciu v podnikoch. Samotný systém ServiceBase implementuje veľkú časť celej ITIL normy a neustále je rozširovaný. Samotný ITIL systém obsahuje dve dôležité časti. Jedna časť je spojená s definíciou služieb a ich vlastností, druhá časť popisuje podporu služieb a problémov spojených s ich behom.

Service delivery (Tactical management)

- *Service level management**
- *Capacity management*
- *Availability management (dostupnosť)**
- *Contingency planning (risk)*
- *Financial management*

Service support (Operational management)

- *Configuration management**
- *Service desk**
- *Incident management**
- *Problem management**
- *Change management**
- *Release management**

Časti, ktoré sú označené hviezdičkou sú implementované systémom ServiceBase. Modulov, ktoré obsahuje ITIL nie je príliš mnoho. Moduly sú však veľmi rozsiahle. Samotné doporučenie ITIL verzie 3 je zložené z niekoľkých kníh. Výsledné riešenie je teda istým výberom z možných riešení a úprava do bežnej praxe.

2.4 Moduly systému ServiceBase

Systém ServiceBase je navrhnutý ako webová aplikácia, ktorá umožňuje dopĺňať nové modely. Architektúra systému je teda otvorená a je možné implementovať nové časti s využitím doterajších existujúcich riešení. V súčasnej dobe tento systém obsahuje nasledujúce moduly:

- *Incident management*
- *Problem management*
- *Service level management*
- *Configuration management*
- *Knowledge base*
- *Change management*
- *Release management*
- *Reports / Statistics*
- *Workflow management*
- *System configuration*
- *User management*
- *Notifications & events*
- *HRE & CRM*

Okrem toho samozrejme obsahuje moduly, ktoré zabezpečujú správu užívateľov a správu údržby samotného systému. Súčasťou modulov sú aj notifikácie, teda upozornenia na význačné udalosti (založenie nového incidentu, servisných požiadaviek, zmeny stavov atď.), ktoré vznikli a je potrebné ohlásiť ich povereným osobám. Dôležitou súčasťou sú i listy a stromy pre zobrazovanie množstva dát z databázy. Tieto umožňujú i bežné operácie ako mazanie, exporty do definovaných formátov a tlač. Mimo funkčnosť, ktorá je dôležitá pre zákazníkov je zaradený do systému aj konfiguračný modul, pomocou ktorého sú nastavované najdôležitejšie parametre pri inštalácii.

2.4.1 Incident management

Incident – udalosť (krátkodobý jav), ktorá sa vyskytne v istom systéme (nefunkčná tlačiareň, monitor, účtovnícky server, nebeží web server).

management tvorí najdôležitejšiu časť pre zákazníkov, ktorí systém používajú. Incident management je modul, v ktorom sú zapisované všetky udalosti a chyby vyskytujúce sa v IT infraštruktúre zákazníka. Pre zákazníka je toto dôležité, pretože pri výskyte takýchto udalostí je ich možné ihneď registrovať, predávať k riešeniu a následne pokiaľ sú vyriešené, tak nasadiť a odstrániť chyby.

Súčasťou incident management je i správa tzv. servisných požiadaviek. Servisná požiadavka je žiadosť o pomoc, prípadne asistenciu. Pokiaľ incident je výlučne chyba, nefunkčnosť časti, servisné požiadavky nemajú s nefunkčnosťou nič spoločné. Servisnou požiadavkou môže byť žiadosť o výmenu toneru v tlačiarni.

Samotný modul incident management nie je však vytvorený len pre evidenciu incidentov a servisných požiadaviek. Okrem evidencie je teda umožnené riešiť konkrétne incidenty a požiadavky.

Toto sa deje za pomoci mechanizmu prepínania stavov podľa definovaného poradia (tzv. *workflow*). Na základe súčasného stavu môže konkrétny riešiteľ meniť stav a predávať k riešeniu incidenty a požiadavky iným riešiteľom, skupinám riešiteľov a rolám. Pokiaľ je incident vyriešený riešiteľom, je prepnutý do definovaného stavu a čaká na overenie. Ak je riešenie dostatočné a správne, potom je uzavretý a považovaný za vyriešený. Takýto mechanizmus správy umožňuje potom zákazníkom veľmi rýchle odozvy na udalosti, ktoré sa vyskytnú. Zároveň je modul incident management prepojený s modulom štatistík, ktorý poskytuje informácie a prehľady napr. o počtoch incidentov, požiadaviek, ich dobe riešenie atď. Modul incident management je prepojený i s modulom znalostnej bázy (knowledge base) a service level management (SLM) modulom.

Incident management využíva modulu pre notifikácie. Pomocou tohto modulu upozorňuje na zmeny stavov (napr. formou emailov), výskyt nových incidentov atď. V prípade, že incident nie je do nastaveného času (preberá sa zo služby) vyriešený, sú vyvolávané tzv. eskalácie a užívatelia, ktorí majú riešiť tento incident sú upozorňovaní o neplnení svojej povinnosti.

Prepojenia modulu incident management:

Incident management -> Service level management
Samotný incident vždy musí obsahovať službu, v ktorej sa incident vyskytol. Služba je súčasťou incidentu. Z modulu SLM sú potom vyberané časy, behu služby a časy, za ktoré musia jednotliví riešitelia, úrovne riešiteľov zareagovať a vyriešiť incidenty.

Incident management -> Knowledge base
Do znalostnej bázy je prevedený export riešenia incidentu, pokiaľ sa vyskytne incident s podobnými príznakmi, je možné sa obrátiť na existujúce riešenie a toto využiť.

Incident management -> Workflow management
Niekedy je nutné schváliť požiadavky na nové zariadenia, preto je zavedený mechanizmus schvaľovania – je možné schvaľovať iba servisné požiadavky.

Incident management -> Configuration management
Z tohto modulu sú preberané prvky, na ktorých sa vyskytla chyba. Prvok nemusí byť pri zadávaní vybraný, preto sa niekedy nevyskytuje.

Incident management -> Problem management
Je možné exportovať incidenty, ktoré v konečnom dôsledku sa prejavujú často a nie sú teda už krátkodobého (incident), ale dlhodobého charakteru (problém).

2.4.2 Problem management

Problém – dlhodobý jav, ktorý pretrváva (rozdiel oproti incidentu, ktorý je krátkodobý – niekoľko často sa vyskytujúcich incidentov tvorí potom už problém v systéme).

Modul problem management je taktiež z pohľadu zákazníkov dôležitý modul. Poskytuje rovnaké nástroje ako incident management s výnimkou schvaľovania. Pri zadávaní problémov nie je povinné navyše zadanie služby. Rovnako problémy môžu vznikať exportom z incidentov. Samotné zmeny stavov pri riešení incidentov a problémov sú mierne odlišné.

2.4.3 Service level management

Modul ktorý tvorí základ systému a helpdesku. Službu je možné chápať ako IT službu (tlačové služby, aplikácie účtovníctva, kancelárske nástroje, webové aplikácie s ktorými užívatelia pracujú). Existuje pritom skoro vždy väzba medzi službami samotnými. Príkladom je vzájomná previazanosť fungovania tlačových služieb a služieb sieťových. Pokiaľ nefunguje správne sieťová služba v podniku, kde v sieti je zapojená často sieťová tlačiareň, prestane zároveň so sieťovou službou fungovať tlačová služba. Systém ServiceBase odlišuje dva typy služieb:

- *Služba technická*
- *Služba obchodná (tzv. Byznys služba)*

Technické služby sú založené prevažne na hardvérových prostriedkoch (sieť LAN, fyzické zariadenia – servery atď.). Naproti tomu obchodné služby sú tie, s ktorými pracujú potom bežní užívatelia v infraštruktúre IT podniku. Príkladom obchodnej služby môže byť eshop, účtovnícky software, systém výroby, poštová služba (beží na poštovnom serveri – technická služba).

Okrem iného je nutná väzba medzi technickými a obchodnými službami. Pri tejto väzbe technická služba podporuje beh obchodnej služby. Samotné obchodné služby majú medzi sebou taktiež väzbu. Obchodné služby sa môžu vzájomne podporovať vo svojom behu.

Každá zo služieb v katalógu obsahuje samozrejme zoznam užívateľov, ktorí sú za túto službu zodpovední. Základné pozície sú manager a metodik (prípadne metodici) služby. V zozname služby sa vyskytujú taktiež užívatelia, ktorí tvoria samotnú podporu služby v prípade problémov. Sú to jej riešitelia. Tí sú rozdelení až do troch úrovní a takto dokážu byť predávané incidenty na niekoľko úrovní užívateľov.

Súčasťou definície služby samotnej sú jej časy behu. Čas behu vymedzuje obdobie, kedy je služba dostupná. Niektoré služby bežia iba v pracovnú dobu zamestnancov, iné sú prístupné nepretržite. Tieto časy sú dôležité pri vypočítavaní eskalácií incidentov a problémov. Samotné eskalačné časy ešte využívajú tzv. reakčné doby a doby riešenia služby. Tieto sú kategorizované podľa priority s akou je incident zadávaný. Kombináciou týchto elementov vznikne potom výsledný čas, po ktorom budú rozposielané notifikačné emaily a správy signalizujúce nečinnosť riešiteľa.

2.4.4 Configuration management

Je modul spravujúci systémové prostriedky zákazníka. Pokiaľ má zákazník bežiace IT služby vo svojom systéme, potom tieto vyžadujú isté technické prostriedky pre svoj beh. Okrem iného napr. každý počítač alebo iné zariadenie, ktoré je v infraštruktúre využívané je taktiež súčasťou IT služieb, teda i súčasťou prostriedkov zákazníka. Práve pre správu týchto technických prostriedkov je určený modul konfiguračného managementu (v systéme ServiceBase nazvaný ako CMDB).

Samotný prvok je evidovaný v databáze. Okrem informácií, ako sú meno, typ, popis, je možné evidovať i sériové čísla alebo vlastníkov jednotlivých prvkov. V prípade problému je takto ľahké zistiť zodpovednú osobu alebo i servis a kontaktovať ich. Licencie sú evidované taktiež. Tieto sú dôležité pri software, kedy niektoré majú obmedzenú dobu platnosti alebo podpory a je ich potrebné obnoviť.

Organizácia prvkov v podniku však musí byť istým spôsobom štruktúrovaná. Rovnako ako v realite sú servery rozložené v miestnostiach na to určených, musí byť táto skutočnosť prenesená i to evidencie konfiguračného managementu. Systém ServiceBase zavádza preto umiestnenia prvkov. Takto je možné definovať miestnosti a oblasti, a priradiť k ním prvky, ktoré sa v nich nachádzajú.

Prvky ako také sú ešte navyše organizované do hierarchií rovnako, ako služby. Jednotlivé zariadenia môžu byť zložené a závislé na iných prvkoch (napr. server – záložný zdroj). Okrem toho niektoré prvky nemusia byť aktívne a momentálne vyradené alebo nedostupné. I toto je súčasťou CMDB modulu.

2.4.5 Knowledge base

Znalostná báza systému ServiceBase. V tomto module sú ukladané akékoľvek znalosti, ktoré môžu prispieť k urýchleniu riešenia problémov alebo incidentov pri správe infraštruktúry. Pokiaľ existuje riešenie v znalostnej báze, je vhodné ho použiť a aplikovať na riešenie.

Údaje do knowledge base sú prenášane cez export alebo vytvorením novej znalosti. Ku každej je evidované riešenie, ktoré bolo prevedené a popis problému alebo incidentu. Znalosti sú prehľadne evidované v kategóriách znalostí. Kategórie si môže vytvárať každý zákazník individuálne. Samozrejmosťou sú bežné operácie, ako exporty do iných formátov a tlač.

2.4.6 Change & Release management

Zámer change manegementu je veľmi široký. Môže obsahovať všetky zmeny, ktoré budú mať pravdepodobne dopad na IT služby. Požiadavka na zmenu môže byť napr. zmena práv, zmena HW a SW vybavenia, konfigurácie a iné. Oproti release managementu má omnoho širší záber. Evidencia požiadaviek v systéme ServiceBase obsahuje i schvaľovacie mechanizmy a posúvanie požiadaviek jednotlivými fázami. Dokumenty, ktoré sú vytvorené tak menia svoj stav podľa fázy, v ktorej sa práve nachádzajú.

Release management je omnoho užšie zameraný systém. Nie všetky požiadavky na zmenu končia vydaním tzv. release. Release management je zameraný predovšetkým na softwarové zmeny, ktoré sa uskutočňujú. Napr. pokiaľ ide požiadavka na zmenu istej časti systému, je možné že po implementácii prejde do release managementu a odtiaľ bude inštalovaná zmena do softwaru.

2.4.7 Statistics

Štatistiky sú dôležité zo strany zákazníka a jeho manažmentu. Veľmi často požadujú nadriadení pracovníci výkazy o práci od svojich zamestnancov. Pomocou štatistík si môžu jednoducho získať prehľad o činnostiach zamestnancov pri riešení problémov a incidentov v systéme. Štatistiky sú prevažne upravované špeciálne pre každého zákazníka osobitne. Vždy však obsahujú prehľady o uzavretých a riešených incidentoch alebo reakčné doby riešiteľov, prípadne časy riešenia. Tieto doby je možné vytvárať špeciálne podľa zadávateľov a podobne. Zároveň možno zaviesť štatistiky podľa služieb, ktorá mala najviac porúch a výpadkov.

Štatistiky v systéme ServiceBase bývajú veľmi často upravované na prianie zákazníka. Preto existuje len definované rozhranie a doplnenie novej štatistiky zväčša prebieha potom iba na databázovej strane. Takéto všeobecné riešenie nesie niekoľko výhod oproti pevne implementovaným riešeniam. Štatistiky je samozrejme možné tlačiť a exportovať do zvolených formátov.

2.4.8 Workflow management – schvaľovanie

Modul určený k evidencii schvaľovaní pri servisných požiadaviek. Umožňuje prehľadnú evidenciu a schvaľovateľom i schválenie čakajúcich servisných požiadaviek. Schvaľovanie je možné previesť i v zastúpení, kedy zástupca schvaľovateľa schváli požiadavku.

2.4.9 System configuration

Konfigurácia samotného systému ServiceBase je taktiež dôležitá. Je možné nastaviť niekoľko parametrov a ovplyvniť tak značnou mierou jeho chovanie. Systém ma tieto možnosti nastavenia:

Notifikácie udalostí
Nastavenie jednotlivých upozornení vyvolaných udalosťou. Na výber je niekoľko udalostí, na ktoré je možné zareagovať a poslať upozorňovací email. Príkladmi sú zmeny stavov, zmeny priradenia riešiteľov, eskalácie, vytvorenie nových objektov (incident, problém, užívateľ, atď.). Notifikácie je možné aktivovať / deaktivovať podľa potreby.

Voliteľné polia
Súčasť systému, kedy každý zákazník si môže vytvoriť k záznamom doplňujúce políčka, ktoré sa budú zobrazovať pri editovaní alebo zakladaní nových objektov. Ku každému voliteľnému poľu je potrebné jeho meno, dátový typ, popis a povinnosť. Tieto polia sa bez zásahu programátora nebudú zobrazovať vo výstupoch.

Číselníky
Sú možnosti, ktoré sú ponúkané užívateľovi v istých políčkach formulárov. Existujú definované číselníky, ktoré sú vedené v systéme pevne. Existujú však číselníky, ktoré sú definovateľné rovnako, ako voliteľné polia, užívateľom.

Jazyky
V tejto časti si zákazník môže definovať vlastný jazyk. K nemu potom musí doplniť všetky lokalizácie pre každý popis, ktorý je v systéme použitý.

Globálne parametre
Táto súčasť obsahuje niekoľko parametrov pre nastavenie systému. Sú nimi napr. základné hodnoty pre eskalačné časy, reakčné časy, časy behov služieb. Tieto hodnoty budú použité ako predvolené, pokiaľ užívateľ pri zakladaní nevyplní svoje. Okrem iného je možné zmeniť veľkosť listov, predvolené skupiny riešiteľov, či označenie skupiny operátorov.

Aktuality
Umožňuje doplniť napr. informácie o zmenách v systéme, prípadne iné, ktoré sa zobrazujú potom ihneď po prihlásení do systému. Aktuality sa budú zobrazovať podľa dátumov a iba počas svojej doby platnosti. Aktuality môžu byť využité na upozornenie o výpadkoch služby.

Emaily
Je jednoduchý zoznam emailov, ktoré boli odoslané užívateľom (napr. notifikačné emaily systému).

2.4.10 User management

Účelom modulu je evidencia všetkých užívateľov systému, ktorí sú potrební pri práci. Užívatelia v systéme potom vystupujú v iných moduloch. Môžu byť použití ako zadávatelia, riešitelia, manageri, metodici alebo operátori. Pokiaľ užívateľ nie je v tomto module, nie je možné ho ani vybrať v iných moduloch.

Užívatelia systému sú rozdelení do jednotlivých užívateľských skupín a rolí. Každý užívateľ môže byť členom niekoľkých rolí a skupín. Samotné skupiny majú potom ešte definované role. Prvky ako role a skupiny potom majú svoje práva. Základným prvkom práv v systéme je rola. Rola má presne definované svoje rozsahy a časti systému, do ktorých má prístup a akým spôsobom (vytváranie,

mazanie, iba prehliadanie). Nasleduje skupina, ktorá má pridelené role a definuje napr. skupinu operátorov a ich kompetencie pomocou rolí. Užívateľ môže mať priradené role i skupiny. Práva z rolí a skupín sa mu vo výsledku sčítajú (pokiaľ nemá priradenú priamo rolu pre prístup, ale má ju definovanú skupina, potom je to rovnaké, ako priradenie role priamo užívateľovi).

2.4.11 Notifications & Events

Udalosti sa v systéme ServiceBase vyskytujú pomerne často. Samotná udalosť je napr. založenie nového incidentu. Reakciou na túto udalosť je odosielanie notifikačných emailov a správ kompetentným osobám, aby došlo k ich upozorneniu na vznik nového incidentu. Títo potom majú vyhradené časy, za ktoré musia zareagovať na vytvorenie incidentu. V tomto čase sú im odosielané rovnako emaily o krátení sa času na reakciu a riešenie. Toto sú ďalšie typy udalostí. Sú vyvolávané systémom samotným. Iným typom udalosti a notifikácie je zmena stavu takéhoto incidentu. Pokiaľ priradí operátor k incidentu riešiteľa, tento riešiteľ mení stavy podľa aktuálnej situácie. Pokiaľ incident vyrieši, označí ho zmenou stavu ako dokončený a operátorovi sú odosielané emaily o dokončení riešenia. Takýmto spôsobom je zabezpečená informovanosť kompetentných osôb pomocou systému notifikácií v ServiceBase.

2.4.12 HRE & CRM

Moduly sú určené pre interné použitie firmou Crux IT s.r.o. Modul HRE je určený pre personálne oddelenie, ktoré sa zaoberá sprostredkovaním práce uchádzačom. Pre nich je tento modul i nastavený. Obsahuje zoznamy uchádzačov a voľných pracovných pozícií. Zoznamy firiem, ktoré poskytujú voľné pracovné miesta. Okrem iného spolupracuje tento modul s webovou prezentáciou firmy, kde sa potom zobrazujú voľné pozície pre uchádzačov a môžu vypĺňať a odosielať svoje žiadosti a ponuky.

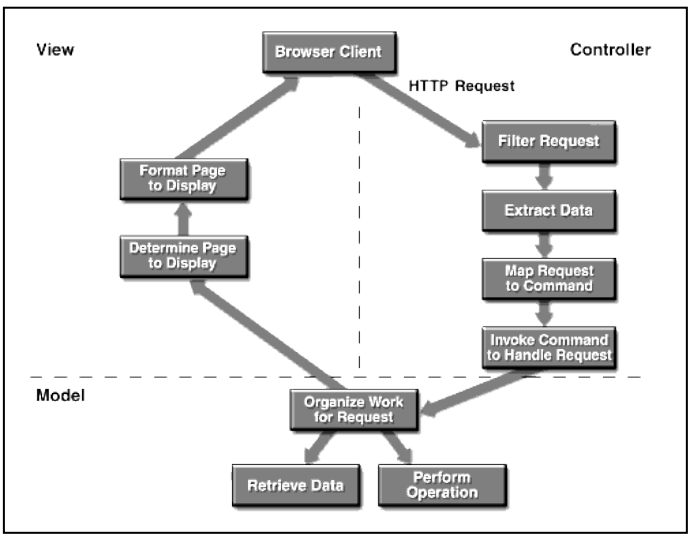
Modul CRM je určený pre obchodné oddelenie firmy Crux IT s.r.o. Umožňuje evidenciu zákazníkov firmy a monitorovanie obchodnej činnosti a ponúk na spoluprácu. Týmto uľahčuje prácu obchodnému oddeleniu. Poskytuje v súčasnej dobe iba základné možnosti evidencie firiem a ich ponúk. V budúcnosti bude možnosť rozšíriť moduly o ďalšie funkcie, ktoré vyplynú z potrieb oddelení firmy.

Oba tieto moduly sú súčasťou systému ServiceBase, ale nevyužívajú iné moduly okrem modulu user management. Na tieto dva moduly nie sú naviazané ani iné moduly. Využívajú len základnú architektúru ServiceBase. Pre zákazníkov navyše nie sú tieto moduly vôbec prístupné.

2.5 Architektúra ServiceBase

Systémová architektúra je navrhnutá ako štandardná trojvrstvová. Obsahuje teda tri základné časti ako prezentačnú, logickú a databázovú vrstvu. Pre prezentačnú vrstvu je použitá technológia servletov v kombinácii s *java server pages* (inak tzv. JSP). Servlety nie sú použité v pôvodnej forme, ale je framework je mierne modifikovaný tak, aby boli čiastočne prekryté zložité kódy. Okrem iného takáto modifikácia bola nutná z hľadiska overovania prístupu k jednotlivým stránkam. Nad každou stránkou sa takto nachádza mechanizmus práv, ktorý blokuje neoprávnené požiadavky. Servletová technológie je založená na klasickom návrhovom vzore *model – view – controller* (skratka MVC). Modelom sú použité dátové objekty a metódy logickej vrstvy, view je konkrétny jsp súbor a controller je logika na pozadí stránok. Controller riadi celý tok spracovania stránky. Vie presne, ktorá stránka alebo trieda má byť vyvolaná na základe konfigurácií. V systéme ServiceBase je toto realizované pomocou tzv. *action-config* súborov. Tieto sú jednoduché XML súbory, v ktorých sú definované stránky

a odpovedajúce triedy. Obr. 12 vystihuje presne architektúru MVC v prezentačnej vrstve systému ServiceBase. Získavanie dát pre Model je zaistené pomocou logickej a databázovej vrstvy systému.



OBR. 12 MVC PRE SERVLETY

Logická vrstva systému je postavené na hierarchii rozhraní a tried. Rozdelená je navyše podľa modulov. Každý modul má svoju časť logickej vrstvy a volaním metód je možné získavať dáta. Vrstva zabezpečuje okrem iného i správne prepínanie stavov, odosielaní notifikácií alebo vyvolanie udalostí. Základ systému tvorí niekoľko rozhraní, ktoré sú v hierarchii a dedia navzájom. Trieda implementujúca rozhranie potom musí implementovať všetky metódy nadradených rozhraní. Logická vrstva je prvou vrstvou, s ktorou spolupracuje model. K databázovej vrstve musí pristupovať každá trieda vždy cez logickú vrstvu.

Databázová vrstva je postavená na štandardnom java database connectivity. Podporované sú databázy Oracle a MSSQL. Konkrétny ovládač je vždy nastavený v konfiguračnom súbore. Cez tento ovládač sú potom posielané požiadavky v dialekte konkrétnej databázy. Rozdiely sú riešené zavedením špeciálnej triedy, ktorá obsahuje rozdielne názvy funkcií a zmeny medzi typmi databáz. Pri konštrukcii dotazu autor musí vedieť, že tieto dve databázy majú iný názov funkcie. Databázové metódy sú volané vždy iba z logickej vrstvy.

2.6 Refaktoring architektúry systému

Systém ServiceBase je relatívne starý. Vyvíjaný bol postupne na základe zvyšovaných nárokov od zákazníkov. Postupom času boli doplnené nové moduly, ktoré predtým neexistovali. Zároveň boli časti rozšírené o nové funkčnosti. Na vývoji pracovalo v priebehu času množstvo vývojárov. Samotná architektúra v pôvodnom návrhu nemohla počítať s rozsiahlymi rozšíreniami a zmenami. Bola navrhnutá ako jednoduchý systém, ktorý bol predovšetkým určený pre menších zákazníkov.

Najväčšie zmeny v architektúre prebiehali v služieb a incidentov. Tieto časti boli modifikované veľmi často a postupom času sa v kóde celého systému začali vyskytovať problémy súvisiace s nie celkom správnou implementáciou. Preto celý mal prejsť refaktoringom. Druhým dôvodom pre refaktoring bol aj celkový výkon systému a jeho modulov. Modul služieb bol napr. veľmi pomalý. Pri niektorých stránkach boli zbytočne načítané zoznamy obsahujúce veľké počty záznamov na stránku. Z týchto zoznamov bolo potrebné však vyfiltrovať iba potrebnú veľmi malú časť. Toto zaťažovalo nielen samotnú prezentačnú vrstvu systému ale i spojenie medzi databázou a databázu samotnú.

Prenos dát bol veľmi veľký, čo má za následok zbytočné réžie a spomalené odozvy. Refaktoring takýchto miest mal priniesť významnú zmenu.

Podľa pravidiel refaktorovania nemohol systém prejsť naraz celkovou úpravou. Preto boli zmeny realizované iba na kritických miestach. Prvým miestom boli služby. Ako najpomalšie miesto systému bola teda zmena najnutnejšia.

Pri refaktorovaní služieb boli odhalené nasledujúce nedostatky:

a. *Častý opakovaný prístup do databázy*

Modul veľmi často nevyužíval už načítané zoznamy, ale opakovane pristupoval do databázy a čítal ich znovu niekoľkokrát.

b. *Zlé filtrovanie údajov*

Počas spracovania boli načítané celé zoznamy dát z databázy a prechádzané v cykloch, aby bolo vyfiltrovaných len niekoľko položiek (tieto miesta sa opakovali hneď niekoľkokrát – navyše prevádzali veľmi podobné činnosti). Toto pôsobilo veľkú réžiu na spojenie i záťaž aplikačného serveru.

c. *Načítanie dát z requestu*

Celé načítanie a spracovanie prebiehalo vrámci jednej metódy. Táto potom v službách presahovala niekoľko stoviek riadkov kódu a tvorila neprehľadné úseky.

d. *Nesprávne pomenovanie premenných*

Premenné nemali dostatočne výstižné názvy a bolo ťažké orientovať sa v takomto kóde. Jeho význam a spôsob práce nebol ani komentovaný. Mimo iného sa často krát opakovali tie isté úseky.

e. *Veľké množstvo už nepoužívaného kódu*

Dlhé úseky kódu boli staré a nefunkčné. Dáta z databázy však neustále čítali a tvorili veľkú réžiu.

f. *Použitie „magických" čísiel pri vetvách if*

Tieto čísla boli okomentované, no opakovane používané, čo spôsobovalo potom problém pri zmene hodnoty.

g. *Celkovo zlý návrh riešenia modulu služieb (výskyt i v ostatných moduloch)*

Triedy, ktoré spracovali služby (napr. vytvorenie novej, editácia, duplikovanie) netvoria žiadnu hierarchiu a nemajú žiadne spoločné metódy. Pri zaslaní požiadavky na spracovanie však rozdiel pri spracovaní novej služby a editovaní je minimálny. Preto z pohľadu návrhu by mala existovať súvislosť. Tá však neexistovala. Pri načítaní novej služby z requestu a načítaní existujúcej bol kód kopírovaný z jednej triedy do druhej. Duplicita, ktorá vznikla pôsobila problémy, keďže nebolo možné udržiavať stovky riadkov kódu vo všetkých triedach naraz a poznať drobné detaily a zmeny. Tento problém je jedným z najväčších v systéme.

h. *Volanie databázových metód na prezentačnej vrstve*

Prístup k databázovej vrstve priamo odporuje trojvrstvovej architektúre. Volania by mali byť výlučne na logickú vrstvu, ktorá ďalej spracuje údaje potrebným spôsobom. Takto dochádzalo k nedorozumeniam a častým chybám. Príkladom je vytvorenie nového ID pre službu – nové ID by správne malo byť priradené tesne pred uložením, teda v logickej vrstve. To však bolo načítané prvý krát už na prezentačnej vrstve, priradené službe, neskôr na logickej a dokonca i databázovej vrstve. Problém sa prejavil tak, že ID novej služby bolo vždy o tri väčšie ako predošlej.

i. *Atribúty requestu nemajú pevné pomenovanie*

Napr. atribút služby, ktorý drží meno by mal byť nazvaný rovnako pri novej službe a rovnako pri editácii (vyskytovali sa pomenovania napr. SERVICE_NAME a ServiceName). Maximálne by ich mal odlišovať len prefix, ktorý by vyjadril príslušnosť k operácii pre ľahšie

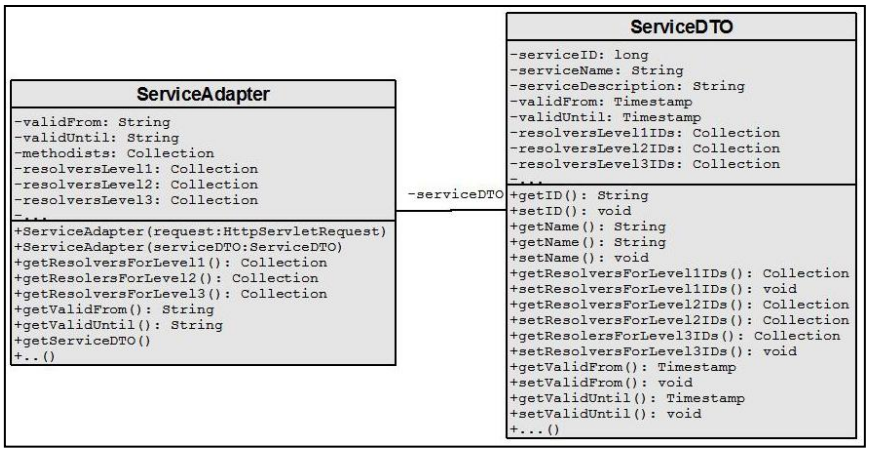
odlíšenie. Prefix nie je odlišnosť potom a zjednotenie načítania údajov z requestu by bolo jednoduché (pomenovanie NEW_SERVICE_NAME a EDIT_SERVICE_NAME je po zmazaní prefixu SERVICE_NAME).

j. Dátový objekt (ServiceDTO) obsahuje String a Timestamp atribúty

Vznik týchto atribútov bol podmienený konverziami. Pokiaľ .jsp súbory potrebujú pre zobrazenie dátumových položiek hodnoty typu *String,* databáza potrebuje položky dátumu výlučne vo formáte *Timestamp*.

Najzávažnejšími chybami sú duplicity a nesprávne volanie metód databázovej vrstvy. Tieto však nemôžu byť odstraňované pokiaľ existujú ešte nezrovnalosti menšieho charakteru. Preto sú na začiatku odstránené chyby typu *d, f, i* a *h*. Po ich odstránení dôjde k zjednodušeniu. Ihneď na to je možné odstrániť chyby typu *a, b* a *c* (v poradí, akom sú uvedené). Ako posledné je možné potom previesť odstránenie nedostatku *g* a následne i *j*. Toto poradie je overené praktickým skúmaním. Ak by bol prevedený pokus o odstránenie nefunkčného kódu skôr ako premenovania a odstránenia drobných nedostatkov, vedie to prevažne k neúspechu refaktoringu. K neúspechu toto riešenie vedie najmä neprehľadnosť a ťažká čitateľnosť kódu. Programátor len veľmi nepresne môže určiť a odhadnúť, ktoré časti sú už nepoužívané. Pri správnom prístupe už počas drobného refaktorovania môže robiť poznámky, ktoré časti sú podozrivé ako už nefunkčné.

Riešenie zlého návrhu je celkom jednoduché. Jediná podmienka pre uplatnenie refaktoringu a úspešné nasadenie je zjednotenie názvov jednotlivých premenných na stránke a uplatnenie ostatných refaktorovaní pre zlepšenie prehľadnosti kódu. Programátor by už za chodu prišiel na jednoduché riešenie. Načítanie údajov z requestu napr. premiestniť do osobitnej metódy. Ďalším krokom je organizácia dát podľa skupín objektov. K službe napr. patria reakčné časy, doby reakcií atď. Tieto sú ale ukladané ako osobitné dátové objekty a mali by aj ich načítania byť zhlukované. Postupne tak budú vytvorené skupiny. Z týchto skupín sú vytvorené menšie metódy. Ďalšie kroky delenia na metódy sú z praktického hľadiska zbytočné. Príliš mnoho malých metód komplikuje situáciu, ak nie sú opakovane využité i v iných častiach. Nakoniec je možné po refaktorovaní tried pre uloženie novej služby a editovanej služby spraviť zlúčenie rovnakých častí. Výhodný môže byť adaptér alebo využitie vytvorenia hierarchie. Triedy pre uloženie novej a editovanej služby získajú nového spoločného predka. Tento predok bude abstraktná trieda a bude dediť z triedy, z ktorej dedili predtým pôvodné triedy. Tie následne zdedia z novej triedy. Implementácia rozhrania *IAction* ostane nezmenená. Metóda *execute* tak ostane v každej triede ako doteraz. Toto je jeden spôsob riešenia problému. Iný prístup je použitie adaptéru, ktorému predá programátor request objekt, ktorý obsahuje údaje. Tento adaptér spracuje údaje a vytvorí objekt a poskytne pomocou *get* metódy. Výhodou riešenia je aj vyriešenie problému s prevodmi dát medzi formátmi. Adaptér môže poskytovať metódy na získanie dátumu pre .jsp súbory. Zároveň pri zostavovaní objektu pre databázu ale vyplní dáta správnym formátom. Pre použitie adaptéru v tomto prípade je aj jeho znovupoužiteľnosť v iných prípadoch. Adaptér môže byť navyše použitý i v súvislosti s databázovou vrstvou. Po načítaní dát z logickej vrstvy môže byť výsledok prenechaný adaptéru, ktorý ho pripraví na zobrazenie vo vybranom .jsp súbore. Dátový objekt, ktorý obsahoval napr. pre časové položky ako hodnotu vo formáte *String* i *Timestamp* tak môže byť upravený. Vynechané budú hodnoty typu *String.* Odstrániť je potrebné aj napr. atribúty, ktoré sú *Collection*. Pre databázu postačuje kolekcia ID objektov. Pre zobrazenie je potrebné poznať i mena objektov a doplňujúce údaje. Tieto môžu byť presunuté do adaptéru. Dátový objekt bude ďalej zjednodušený. Takto bude odstránená posledná chyba modulu. Výsledné možné riešenie je vidno na Obr. 13.



OBR. 13 VÄZBA ADAPTÉR SLUŽBY A DÁTOVÝ OBJEKT SLUŽBA

Dátový objekt služby je súčasťou adaptéru, ale je prístupný iba cez volanie metódy *getServiceDTO()*. Potom sú prístupné všetky štandardné atribúty služby. Ak je vyžadovaný prístup ku adaptovaným hodnotám potrebných pre zobrazenie, tieto sú skryté pod metódami adaptéru, ktorý ich automaticky prevádza do požadovaného formátu. Pri vytvorení adaptéru je mu potrebné predať objekt, z ktorého bude vytvárať hodnoty. Pokiaľ mu bude predaný objekt request, bude vytvárať údaje z requestu a nakoniec vytvorí dátový objekt. Ak mu programátor predá objekt serviceDTO, bude vytvárať dodatočné údaje z tohto objektu a samotný objekt vytvárať nebude. Príklad je na ZK. 21.

```
ServiceAdapter sa = new ServiceAdapter(request);
ModulFactory.getServiceManagement().insert(sa.getServiceDTO());
...
ServiceAdapter sa = new ServiceAdapter(serviceDTO);
String validFrom = sa.getValidFrom();
```

ZK. 21 POUŽITIE ADAPTÉRU V SLUŽBÁCH

Podobne ako modul služieb obsahujú veľmi podobné typy nedostatkov i ostatné moduly systému. Refaktoring je potrebné uplatňovať postupne a v definovanom poradí. Menšie celky sú výhodne. Postupný refaktoring zaberie dlhú časovú dobu, ale prínos pre tento systém je veľký. Implementácia nového systému by priniesla rovnaké problémy, preto nie je v momentálnej situácii dôvod na novú implementáciu.

Popri aplikovaní refaktoringu na jednotlivé časti prezentačnej vrstvy by zo systému mala byť odstránená pôvodná bezpečnostná vrstva, ktorá v súčasnej dobe v systéme nezohráva žiadnu rolu. Jej funkčnosť bola nahradená právami v databáze. Prístup k častiam systému nie je teda kontrolovaný v tejto vrstve a môže byť odstránená.

Vo svojom základe po prevedení refaktoringu je systém ServiceBase dobre navrhnutý. Nedostatky je možné odstrániť podobne vo všetkých triedach a moduloch ako v prípade modulu služieb. Prínosom pre systém by bolo použitie technológie ako JPA alebo Hibernate. Ich prínos na databázovej vrstve by bol hlavne čo sa týka podpory databáz. Mohli by odpadnúť prípadné rozdielové triedy, ktoré sa v systéme nachádzajú. Odpadlo by i písanie čistého sql v danom dialekte databázy. V konečnom dôsledku by teda investícia priniesla množstvo výhod. Okrem iného podobné technológie nepracujú potom s *ResultSet* objektmi, ale dokážu do dátových objektov dopĺňať údaje, čo je prínos z hľadiska efektivity práce programátora. Zmenu celej architektúry pre použitie tejto technológie bude možné previesť po aplikovaní refaktoringu a celkového zjednodušenia aplikácie. Momentálne nie je možné

prechod uskutočniť. Na logickú a databázovú vrstvu je po dokončení refaktoringu prezentačnej vrstvy potrebné previezť na správnu formu návrhového vzoru *AbstractFactory* tak, ako je to popísané v časti Refaktoring a návrhové vzory na stránke č. 17.

2.7 Synchronizácia systémov pri vývoji

Požiadavkou firmy Crux Information Technology je súčasná správa starších systémov a nového (napr. systému, ktorý bol refaktorovaný a systému, ktorý má navyše zavedenú technológiu JPA alebo Hibernate, prípadne refaktorovaný systém proti nerefaktorovanému). Staršie verzie systému sú prevažne systémy, ktoré sú v predaji alebo boli už nasadené u zákazníkov. Potom je potrebné mať mechanizmus pri vývoji, ktorý umožní aj zákazníkom rozšíriť funkcionalitu systému o nové prvky a nielen nový systém.

Firma Crux IT využíva pre správu verzií verzovací systém CVS. Tento systém umožňuje operácie systému správy verzií a okrem iného i označovanie súborov vlastnými značkami. Toto je využívané v súčasnej dobe na označenie zmenených súborov v systéme, aby bolo možné kedykoľvek ich vybrať z CVS a vytvoriť z nich patch pre chybu. Okrem tohto využíva firma i systém na správu a logovanie chýb a zmien zvaný JIRA. Do tohto systému je možné priradzovať požiadavky na zmeny, chyby a predávať ich riešiteľom. Ako vývojové prostredie je používané prostredie Eclipse verzie 3.3. Toto poskytuje dostatočné možnosti pri práci s CVS i pri porovnávaní súborov a zmien.

Okrem iného je potrebné vedieť odlíšiť zmenu na novom a starom systéme vzájomne od seba. Ak by boli nasadené zmeny a inštalačné skripty pre nový systém na jeho staršiu verziu, mohlo by to spôsobiť problémy a chyby.

2.7.1 Situácia A (refaktorované systémy)

Riešenie problému vychádza z faktu, že oba systémy sú ekvivalentné, čo sa týka funkčnosti. Obsahujú teda rovnakú funkcionalitu. Jediný rozdiel medzi týmito dvomi systémami je ich vnútorná implementácia. Oba systémy by mali byť refaktorované. Rozdiel medzi nimi je iba technológia databázovej vrstvy. Pri tejto situácii je riešenie nasledovné:

a. *Pre každú zmenu existuje RFC (tzv. request for change) dokument*

Tento dokument popisuje zmenu funkcionality, ktorú je potrebné previesť. K dokumentu môže byť priložená i napr. byznys analýza. Dokumenty je samozrejme potrebné správne verzovať a uchovávať.

b. *V systéme pre logovanie chýb JIRA budú vytvorené požiadavky na zmenu ako pre nový, tak pre staršie verzie systému*

Takto budú vytvorené identifikačné čísla pre starý a nový systém, pod ktorým bude možné nahrať zmenu na CVS a označovať. Požiadavka bude priradená jednému riešiteľovi.

Riešiteľ rieši požiadavku na novom systéme

c. *Po vyriešení prevedie commit a tag na zmenené súbory*

pod číslom, ktoré bolo priradené v systéme JIRA. Riadi sa pritom RFC dokumentom, ktorý bol priložený.

d. *Následne musí riešiteľ previesť zlúčenie zmien do starého systému*

Riešenie je jednoduché pri tejto situácii. Prezentačné a logické vrstvy sú úplne rovnaké, rozdiel je na databázovej vrstve a v dátových entitách. Programátor spraví porovnanie jednotlivých súborov, ktoré upravoval na novom systéme oproti starému riešeniu. Zmeny na prezentačnej (i na .jsp súboroch) a logickej vrstve je možné kopírovať. Zmeny na databázovej vrstve a entitách musí riešiť znovu. Tu neexistuje všeobecné riešenie problému. Jazyk HQL

(JPQL) je mierne odlišný od MSSQL alebo ORACLE. Preto programátor musí použiť pôvodné triedy, ktoré slúžia pre odlíšenie databáz MSSQL a ORACLE. Na potrebné miesta doplniť prípadné rozdiely a zároveň na databázovej vrstve previesť ďalšie zmeny (vytvorenie dotazu). Po prevedení spraví programátor commit a tag na zmenené súbory starého systému pod číslom, aké bolo pridelené v JIRA.

2.7.2 Situácia B (nerefaktorovaný a refaktorovaný systém)

Táto situácia je oproti situácii A komplikovaná. Nakoľko sa odlišuje i prezentačná vrstva medzi systémami, nie je možné zmeny previesť jednoduchým kopírovaním. Vychádzať je možné z bodu c, predchádzajúcej situácie. V tomto kroku je prevedený commit nového systému a jeho zmien. Ďalší bod d. bude odlišný. Nový krok bude nasledovný:

- *Prevedenie porovnania súborov a rozdielov.*
  Programátor porovná aké zmeny sú v súboroch starého a nového systému. Jeho ďalší postup bude ovplyvnený mierou odlišnosti refaktorovanej a pôvodnej triedy.
- Pokiaľ sú zmeny veľké, je programátor nútený zmenu implementovať nanovo. Z hľadiska rizika zavedenia chyby je tento postup výhodný. Programátor môže použiť iba časti kódu, ktoré doplnil kvôli funkčnosti. Ich umiestnenie v pôvodnom systéme bude iné a pravdepodobne budú i mierne zmenené. Časťou súborov, ktoré ostávajú približne v rovnakej forme, sú súbory .jsp prezentačnej vrstvy. Tieto súbory po porovnaní zmien je možné jednoducho doplniť o chýbajúce časti (samozrejme dôsledne s ohľadom na zmeny v pomenovaní premenných a atribútov).

Správa dvoch systémov je v celkovom pohľade veľmi náročná záležitosť. Z hľadiska softwarového procesu je možné udržiavať dokumenty o zmenách a takto mať prehľad o stave. Z pohľadu programátorov a implementácie zmien nie je situácia jednoduchá na riešenie. Problémom ostáva prevedenie zmien v systéme, ktorý nie je refaktorovaný. Zmena v novom systéme spôsobí množstvo iných úprav ako v pôvodnom. Transformácia týchto zmien do pôvodného systému ostáva na miere profesionálnosti programátora. Pretože je táto činnosť veľmi náročná, mali by byť prevedené kroky, ktoré pomôžu nasadiť novú verziu systému u zákazníkov a odstránia problém správy verzií.

Záver

Praktické uplatnenie refaktoringu je náročný proces. Teoretické základy poskytujú základné informácie a návody ako refaktorovať. Tieto sú však jednoduché a v praxi je potrebné vedieť uplatniť všetky pravidlá s úspechom. Systém ServiceBase prešiel a prechádza veľkými zmenami, ktoré majú odstrániť nedostatky, ktoré sa vyskytli. Ich riešenie nebolo jednoduché, pretože komplexnosť systému neumožňuje rozsiahle refaktorovanie celých modulov. Takýto systém vyžaduje osobitný prístup. Postupné drobné zmeny v štruktúre a architektúre priniesli výsledok omnoho rýchlejšie ako plánovaný refaktoring celej časti, ktorý by zaberal týždeň práce programátora. Refaktoring je silnou a efektívnou metódou, pokiaľ ho vie programátor uplatniť. Systém ServiceBase je možné v budúcnosti upraviť, aby bol efektívnejší a flexibilný. Pokračovať možno v prepracovaní databázovej vrstvy na niektorý z frameworkov Hibernate alebo JPA. Okrem iného na prezentačnej vrstve je možné uplatniť framework Java Server Faces, ktorý poskytuje nové možnosti z pohľadu vývoja na prezentačnej vrstve. Po prevedení refaktoringu bude možné systém rozširovať o nové moduly a tak zvýšiť jeho možnosti predaja.

Literatúra

1. Desing patterns and Refactoring. *refactoring.com.* [Online] [Cited: 19 1, 2009.] http://sourcemaking.com.
2. **Martin Fowler, Kent Beck, Vladimír Lahoda.** *Refaktoring: zlepšení existujícího kódu.* - : Grada Publishing a.s., 2003. str. 394. ISBN 8024702991, 9788024702995.
3. **Hoadley, Paul.** en.wikipedia. [Online] 25. 11 2005. [Dátum: 6. 4 2009.] http://en.wikipedia.org/wiki/File:Waterfall_model.png.
4. Designing Enterprise Applications with the J2EE Platform, Second Edition. [Online] [Dátum: 01. 02 2009.] http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/.
5. Java Persistence API. *java.sun.com.* [Online] [Dátum: 02. 02 2009.] http://java.sun.com/javaee/technologies/persistence.jsp.
6. Eclipse Europa Download. *eclipse.org.* [Online] [Dátum: 11. 12 2008.] http://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/europa/winter/eclipse-jee-europa-winter-win32.zip.
7. Hibernate. *hibernate.org.* [Online] [Dátum: 10. 11 2008.] https://www.hibernate.org/.
8. CVS - Concurrent Version System. *cvshome.org.* [Online] [Dátum: 15. 01 2009.] http://www.cvshome.org/.
9. ICEFaces - Open Source Ajax, J2EE Ajax, JSF Java Framework. *icefaces.org.* [Online] [Dátum: 23. 10 2008.] http://www.icefaces.org/.
10. JavaServer Faces Technology. *java.sun.com.* [Online] [Dátum: 23. 10 2008.] http://java.sun.com/javaee/javaserverfaces/.
11. Extreme Programming. *extremeprogramming.org.* [Online] 15. 03 2009. http://www.extremeprogramming.org/.

Prílohy

Diplomová práca neobsahuje prílohy, keďže zdrojové kódy sú vlastníctvom spoločnosti Crux Information Technology s.r.o.